CS352 Lecture - Object-Based Databases

Last revised 10/17/06

Objectives:

1. To elucidate fundamental differences between OO and the relational model
2. To introduce the idea of adding persistence to an OO programming language
3. To introduce OO extensions to the basic relational model
4. To discuss issues involved in choosing an approach for a particular problem

Materials:

1. Excerpt from Industry Trends article - *IEEE Computer* 8/2000 - to read
2. Projectectable of "Relational Model Rest in Peace" cartoon
3. Projectable of Stonebraker matrix

I. Introduction

A. There has been a considerable interest in the last 10-15 years in bringing together two technologies: Object Orientation and database systems.

1. In one sense, this is surprising, because these two technologies have very diverse origins and historically have served diverse application areas.

a) OO: Comes out of the world of discrete simulation. Much subsequent work was motivated by the development of GUI's in the desktop/laptop world. OO is the natural paradigm for designing a GUI and for certain kinds of problems involving objects with complex structure (e.g. computer-aided design).

b) Database systems: comes out of the world of business data processing. Much of the work has been done in the mainframe world. Database systems historically have had a strong batch processing flavor.

2. The desire to bring these two technologies together has arisen from needs in both areas.

a) In the OO world:

(1) The need for PERSISTENCE - retaining objects between runs of a program. (When an object is created by an operation such as `new`, it "lives" only in memory)

Classically, OO systems have used conventional files to store data between program runs. There is a desire in the OO world to take advantage of already developed database technology dealing with such matters as

(a) crash recovery

(b) concurrency

(c) data integrity

(d) security

(e) the ability to use a common database to support diverse applications, rather than having each application area "own" its own data

(f) the ability to query data interactively without having to write a full-blown program to perform the query.

(2) Moreover, there is also a desire to be able to access existing "legacy" data in organizations. Such data is often stored (in large quantities) in relational databases.

b) In the database world

(1) Interest in manipulating data that does not have a simple textual representation (string of characters) - e.g. pictures, sounds, movies, (Note that the "viewing" of non-textual data must be handled in a very different way from the "viewing" of text).

(2) Most of the early work in databases was done in the world of business data processing, which was dominated by mainframe computers and "dumb" terminals. Now, graphical user interfaces and e-commerce are ubiquitous. OO is the natural way to go when creating software in these realms.

B. We saw, in our previous topic, how a "two-tier" or "three-tier" architecture can be used to allow a single system to utilize both an OO user interface and business logic and a relational database. We also saw that this entails explicitly converting information from one form to another, by embedding either dynamic or static SQL in a (typically OO) host language. Thus, programs developed using this paradigm actually make use of two different languages. Moreover, the two languages typically have very different ways of representing data (objects versus simple atomic data)

There is a natural desire to either eliminate the need to use two different languages, or at least reduce differences in data represenations.

C. The task of bringing these two technologies together (in this way or some other way) is rendered non-trivial by some basic differences in the way they handle certain issues, which leads to what some have called an "impedance mismatch" at the boundary between the two systems.

The fundamental concept in OO is the notion of a class, with objects as instances of classes. The fundamental concept in relational databases is the relation (or table), which is a set of tuples (or rows). It would then seem natural to establish the correspondences

class = relation/table
object = tuple/row

But there are crucial problems with this identification:

1. Differences in fundamental notions:

   a) In the OO world, an object has three essential properties: identity, state, and behavior. These are distinct ideas, in the sense that it is conceivable that two objects with different identity might have the same state. Of course, all objects of the same class have the same behaviors.

   Typically, the identity of an object is established by the location in memory where it resides - i.e. when all is said and done an object's identity is basically a memory address. This, of course, is not of any semantic significance, and may even change from run to run of the same program.

   b) What are the essential properties of a table row? Certainly, a row also has identity and state. But - in the basic relational model, rows do not have behavior. More imporantly, though, the notions of identity and state are not distinct: it is inconceivable that a given table might have two rows with same state. [This would violate the set/primary key concept] That is, OO table rows really only have one property: state.

   Thus, the identity of an entity is based on the value of certain of its attributes - something which is semantically meaningful, and doesn't change from run to run of the same program.

   c) This sometimes leads designers of OO classes to incorporate some sort of "key" attribute in order to provide semantically meaningful identify and to guarantee that no two objects have identical state - e.g.

```
class Product
{
     int id;
     ...
```

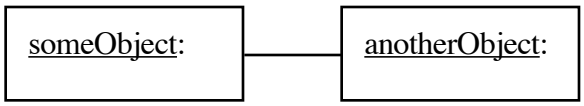
strictly speaking, this is not necessary in an OO system (unless some sort of map is going to be needed to locate the objects, of course) However, something like this is often needed in creating a relational scheme if one cannot guarantee that the remaining attributes of an object will be unique.

d) That is, we have the following:

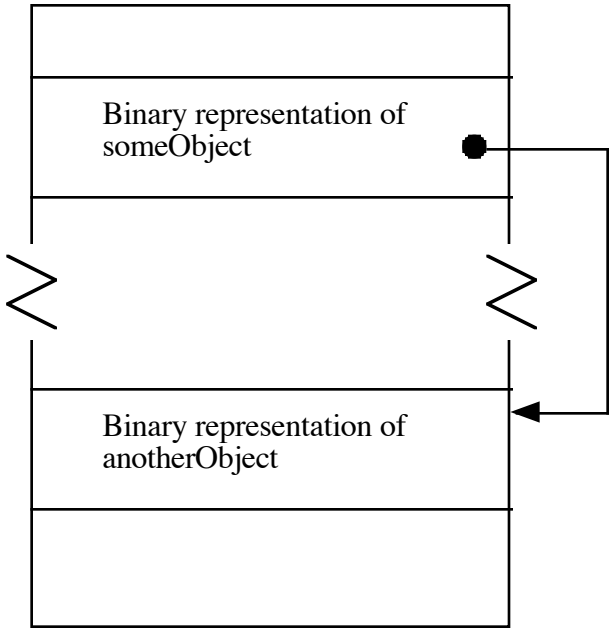
OO:	identity ≠ state
relational:	identity is the same thing as state

2. Related to this is how the two "worlds" handle relationships between objects. Though both can use ER diagrams as design tools, they translate relationships quite differently.

a) In an OO system, relationships (associations) are handled by pointers (called references in Java) - an object that is associated with another object contains a reference to the location in memory where the other object "lives" - e.g.



is represented in memory by



.

b) In a relational system, relationships are handled by foreign keys. Often, relationships are modelled by tables as well - i.e. a relational database table may correspond to an entity (object) or a relationship (association between objects).

c) There are two consequences of this difference:

(1) Foreign keys are semanticaly meaningful; pointers to locations in memory or on disk are not (they're just numbers, and not particularly meaningful ones at that.)

Example: Can you easily tell someone your student ID or SSN if asked?

Can you easily tell someone the location on disk where information about you is stored by the Gordon adminstrative system?

(2) From a performance standpoint, there is a huge difference. One can follow a pointer directly to the object it references; but following a foreign key involves some sort of table lookup. Quantitatively, the time difference may a less than a microsecond versus a few milliseconds - a ratio of over 1000:1.

(3) As an illustration of the issues of efficiency and semantic meaningfulness, consider the following:

Suppose you wanted to find the office of a given faculty member, given their name. You would have to look up the office number (say on the go site), and then go there - something which would take some amount of time and effort. But if you had the office number to begin with, you could go there directly without having to look anything up.

OTOH, as I was revising this lecture two years ago, someone asked me "where do I find room 209?" After playing "20 questions" for a while, I determined that what she was looking for was the secretary's office - at which point giving directions became easy. Like most people, I can remember where a person's office is much more easily than I can remember a room number - when I need to drop something off for the secretary, I never think "I go to room 209".

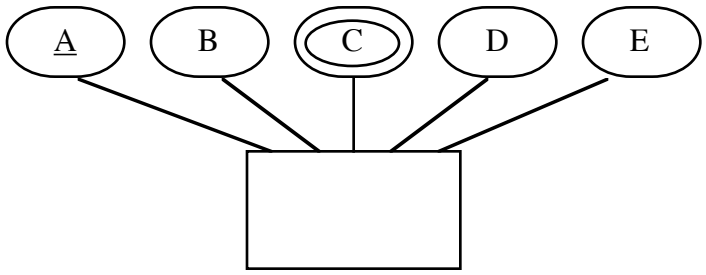
d) That is, we have the following:

OO:	relationship (association) modelled by pointer
relational:	relationship modelled by foreign key

(In this regard, there is more similarity between OO and the network and hierarchical models (which also model relationships by links). However, the latter models are "old technology" and lack the interactive query capability which creates interest in DBMS's in the OO world.)

3. Normalization is an important issue in the relational database world, but not even an issue in the OO world.

   a) Relational databases need to be normalized. Even 1NF precludes non-atomic attributes. We have seen how normalization to 4NF requires us to isolate multivalued attributes in their own relation - e.g.something like



   normalizes to two tables: `(ABDE)` and `(AC)` to satisfy the requirements of 4NF

   b) However, OO classes can contain attributes which are collections of various sorts.

   c) This has some profound implications, which someone has illustrated as follows:

   Is it possible to store a car in standard office filing cabinets?

   ASK

   Yes - if you're willing to take it apart far enough! The problem comes in putting it back together when you need it.

4. The concept of inheritance and related notions (e.g. polymorphism) is handled very differently in the two worlds.

   a) In the OO world, inheritance and polymorphism are two of the three parts of the OO "PIE". Inheritance is specified very simply (e.g. by the Java keyword `extends` or a simple colon in C++). One can access the methods and fields of a subclass without needing to even think about whether they are part of the subclass itself or are inherited from a base class.

b) The concept of inheritance is not part of the basic relational database model. If one uses generalization/specialization in an ER design, one converts this to relational tables in one of two ways:

(1) Create separate tables for each of the specializations. The generalization does not correspond to a table per se - it is created as needed by taking the union of the specialized tables.

(2) Create a table for the generalization that contains all the attributes that are common to all categories; then create a separate table for each specialization that includes the special attributes plus a copy of the primary key of the generalization.

In either case, querying the data requires being conscious of the inheritance - needing either a union or a natural join in queries depending on which way "ISA" has been modelled.

5. Some writers have pointed out that there is also a fundamental philosophical difference between the two approaches to representing information.

a) Database systems seek to foster "data independence" - the idea that data should be stored in such a way as to be independent of the programs that use it - which facilitates sharing of data between application areas and the ability to perform ad-hoc queries.

b) Object-orientation emphasizes the close connection between data and behavior. An object doesn't just incorporate data - it also incorporates methods that operate on the data. Moreover, one of the components of the OO "PIE" is encapsulation, which precludes operating on data except through the set of methods that a class furnishes.

In a generic database, one often needs to use the data in ways that might not have been anticipated when the database was designed. What does one do then? Wait for new methods to be coded? Break encapsulation? Neither seems like a good solution.

D. These issues have led to three different approaches to a solution:

1. Live with the differences. Essentially, this is what many existing systems do. The two-tier and three-tier architectures we discussed earlier are often used with this approach. This means, of course, that the programmer must consciously arrange for transmission of entities between main memory and the database, and must explicitly code transformation between information representations when needed.

2. Create a new object-oriented DBMS model that has all the good features of the relational model (particularly crash recovery, ability to share a common database between applications, and support for ad-hoc queries), but which has the OO class/object as its fundamental concepts.

3. Extend the relational model to make it a better match for the requirments of OO.

II. Approaches to the Development of a new OO Database Model to Replace the Relational Model.

A. Currently, there is no OODB model that is standardized and universally accepted in the way the relational model is; but there are a variety of commercial products and there have been various industry efforts to develop such a model by adding persistence to OO languages.

B. Approaches toward creating an OO data model typically have the following characteristics.

1. The design is based on equation that an entity-set = a class, and the class structure mirrors the class hierarchy and associations in the ER diagram. (Note the similarity between a UML class diagram and an ER diagram - indeed, the latter is one of the ancestors of the former.)

2. The query language is an OO programming language, with extensions to support persistence typically in the form of libraries rather than actual changes to the language. (Contrast this to the use of a separate query language embedded in a host language, such as embedded SQL).

3. An object may be either transient or persistent.

a) Transient objects reside only in main memory, and cease to exist when the program terminates.

b) Persistent objects reside on disk between runs of the program, and a given persistent object may reside either on disk or in memory while the program is running. (But if it resides in memory, it is returned to disk before termination, and may be updated on disk whenever it is changed.)

c) Various OODB models differ as to how the distinction between transient and persistent objects is made.

(1) On a class basis (a class is declared to be persistent, and, as a result, all its instances are persistent) (Persistence by class)

(2) When an object is created (an object can be explicly created as persisent) (Persistence by creation)

(3) An object can be explicitly marked as persistent (by invoking some method) (Persistence by marking)

(4) By reachability from a persistent root. (If an object is persistent, other objects it holds references to also become persistent.) (Persistence by reachability)

4. The act of storing or retrieving data from the database is transparent. Behind the scences, the implementation supports pointers that can either refer to objects that are already present in memory, or to objects that are currently on disk. When a pointer that refers to an object on disk is used, the object in question is brought into memory.

   Thusr, in all cases, apart from initially making an object persistent all other references to the object do not need to treat transient and permanent objects differently. (Contrast this to the use of something like embedded SQL in a host language, where the transfer of information to/from the database is performed by explicit SELECT, INSERT, DELETE, or UPDATE statement written in a different language than the rest of the program.)

C. Example: the book discusses how this might be done in C++. The C++ discussion is actually based on an ODMG extension to C++ that was abandoned in 2002; however, some of the ideas are still in use.

1. A pointer/reference may either contain a memory location, or it may contain a location on disk.

2. The operator overloading mechanisms of C++ are used to allow a programmer to use "pointer syntax" to manipulate an object, regardless of where it actually resides.

   Of course, if an operation is being performed on an object that currently resides on disk, the underlying system must bring the object into memory. This, however, can be done in a way that is transparent to the user; and a technique known as "pointer swizzling" can be used to prevent multiple trips to disk for the same object.

3. The ODMG proposal for C++ also included a library of templates to support referential integrity.

D. The Java extension - known as JDO (Java Data Objects) is currently work in progress.

E. Problems: A lot may be lost by going this way:

1. Support for ad-hoc queries. It would be hard to imagine an ordinary user formulating queries interactively in either C++ or Java! (As a result, some commercial OO DBMS systems provide a SQL query facility as well)

2. Support for "set at a time" processing - to perform some operation on all the members of a collection, one must code a loop using an iterator.

3. Support for referential integrity through notion of keys, etc. (The ODMG extensions put a significant burden on the programmer to keep reference sets and their inverses up to date)

III. Extension/adaptation of the Relational Model to Better Support Use with OO Applications

A. Many of the recent extensions to SQL (in SQL 1999 and 2003) have been concerned with reducing the impedance mismatch between the relational model and the needs of OO systems.

A common thread in many (but not all) of these extensions is the equation domain = class - i.e. entity sets are still relations, but some of their attributes may move beyond traditional restrictions.

B. These extensions move in a number of directions beyond the traditional relational model, including:

1. Allowing non-atomic data types

a) Columns whose values may contain internal structure (i.e. have fields of their own.

The SQL 1999 standard specifies a `create type .. as` facility which allows defining a structured type.

b) Columns which may store a collection (e.g. set, array) of values rather than a single value - a relation nested within a relation. A key motivation here is efficiency: multivalued attributes are associated with multivalued dependencies, which force decomposition during normalization and lead to the need for joins in queries. However, a join is inherently a computationally expensive operation., and transferring

an object containing a collection between memory and the database typically requires a loop that translates between members of a collection and rows in some table arising from normalization.

The SQL 1999 and 2003 standards defined `array` and `multiset` data types. A table column can be defined as some standard type, followed by either `array[`*`some constant`*`]` or `multiset`. What this does, in effect, is to create a table attribute for each row in the table. The DBMS transparently translates between a collection field in an object and a table columns in a database row.

2. Support for complex data types (binary large objects (`blob`) and character large object (`clob`)).

   The DBMS has no knowledge of the internal structure of a large object; it simply allows reading/writing the entire object.

3. Reference data types

   a) System-generated object identifiers (oids)

   b) The ability for a field to store the oid of a row in another table (in effect a pointer to it) rather than a key that must be looked up. Again, efficiency is a key motivation.

4. Support for inheritance

   a) Types based on other types - so that a type includes fields of its own plus fields inherited from a parent type or types.

   b) Tables based on other tables - so that a given row, when inserted into a subtable, also becomes a row in a base table. (This moves into the realm of class = entity-set, of course)

5. Storing procedures in the database along with the data

   a) As methods of specific data types.

   b) As "stand-alone" functions or procedures.

6. Extending SQL to be a more fully-functional programming language in its own right, including various control structures and an ability to call routines written in other languages from within SQL.

   Example: the `CASE` construct in SQL you used in an earlier homework.

7. Various commercial systems developed during the 1990's incorporated some or all of these facilities, which became part of the 1999 SQL standard and the latest revision - SQL 2003 - though standardization of implementations is still a ways off and no vendor comes even close to fully supporting the standard. (And vendors often do things contained in the standard in their own, non-standard way.)

IV. Concluding Thoughts: Where are We Headed?

A. For some time now, interest in the development of a new OO model seems to have been declining..

1. For example, the 3rd and 4th editions of our text had a separate chapter on "Object-Oriented Databases", while the 5th edition combines this with the Object-relational approach to produce a single chapter on "Object-Based Databases".

2. In 2000, an article in *IEEE Computer* made some predictions that seem to have been coming to pass. ("Industry Trends" - *IEEE Computer* 33.8 (August 2000) pp. 16-19.)

   READ marked excerpts from article

3. However, OODBs are quite important in a number of niche markets (e.g. medical information systems), and new developments in this area are always possible.)

B. There is some amount of conflict between those who hold different views about where the OO and database worlds should go.

1. Example: one OO database vendor's Technology Guide begins with a section headed "The computing world has entered a post-relational era".

2. Example: title of a paper "Why the Object Data Model isn't a Data Model"

3. Example: Title of an article: "Good News: The Relational Data Model is Dead". (Actually a critique of OORDMS extensions which - in the author's view - compromise the model.)

C. One tension that often shows up is between efficiency for a specific application, on the one hand, and genericity and modifiability, on the other hand.

1. One writer (C.J. Date) puts it this way: "Although the programming language and database management disciplines certainly have a lot in common, they do also differ in in certain important aspects (of course). To be specific: * An application program is intended - by definition - to solve some specific problem.* By contrast, a database is intended - again by definition - to solve a variety of different problems, some of which might not even be known at the time the database is established." (*An Introduction to Database Systems* - 7th ed (Addison Wesley. 2000) p. 813)

2. It is worth observing that how one intends to use the database affects how one evaluates the relative merits of the traditional relational, object-relational, and OO models.

   a) A strictly OO model has performance advantages for high-volume transaction processing (lots of inserts and deletes, supporting access from many locations simultaneously, e.g. over the web or for an institution with many branch offices.) Joins, in particular, are much more computationally-expensive than following pointers.

   b) The relational model has a mathematical simplicity and elegance that provides better support for preserving data integrity and for ad-hoc queries. When pointers are used to model relationships, more of the burden has to shift to the programmer to navigate them and to ensure that they are manipulated correctly. This gets in the way of facilitating ad-hoc queries by ordinary users.

   c) An object-relational database might allow one to use a relational approach in a domain where the traditional relational model doesn't provide the needed tools, though the question arises as to how much should the basic model be changed without losing the advantages of the relational model.

      Going down this road, of course, raises the issue of "how far do we go?" in terms of compromising the strict relational model to incorporate ideas from OO. Various writers might take a "tigher" or "looser" stance.

      PROJECT: "Rest in Peace" cartoon

3. Arguably, an "OO database" might be more suited for providing persistence in support of a specific application (e.g. CAD), while a relational database might be more suited for storing information in a generic way that supports lots of different ways of accessing it. An object-relational database might be suitable for application domains that demand both the data independence of relational systems and support for complex data.

One widely-cited representation of this approach is the “Stonebraker classification matrix” proposed by Michael Stonebraker of Informix. While this was developed in the 1990’s, I think it still represents a good way of summarizing the issues

PROJECT