CPS352 Lecture - Complex Data Types; Support for Object-Orientation

Last revised February 23, 2023

Objectives:

1. To elucidate problems with the relational model for some applications
2. To introduce OO extensions to the basic relational model
3. To introduce ways to map OO data to the basic relational model

Materials:

1. Projectable of diagram showing OO representations of a relationship by a pointer.
2. Projectable of ER example with multivalued attribute
3. Projectable of Sadalage/Fowler Figure 1.1
4. Projectables of mapping various OO constructs to tables

I. Introduction

A. The relational model has become the dominant database model and is used for most applications.

1. Support for integrating data from multiple applications, and allowing multiple applications to share a common database.

2. An interactive query facility which allows accessing data without custom programming through the use of a common language: SQL.

3. Strong support for ACID transactions that are central to maintaining database consistency in the face of concurrent operations and threats of data lost due to things like hardware failure.

   This is actually not unique to the relational model nor do all relational databases support transactions.

a) Databases based on other models can also offer support for transactions.

b) Not all relational databases support transactions. For example, if transaction support is wanted in mysql, one must specify this when creating the database - otherwise ACID transactions are not supported.

c) However, support for ACID transactions has become recognized as a key feature of relational DBMS's.

B. However, the relational model is designed for structured data - with each row in the table having the same columns, each of which holds an atomic value.

C. Moreover, the relational model creates some difficulties for a system using OO because of what has come to be known as the "impedance mismatch" between OO programming and data storage in relational tables.

II. Unstructured or Semi-structured Data

A. One early addition to the relational model was provision for large unstructured data objects like images or long text documents (perhaps stretching over multiple pages)

1. The SQL standard has added the data types BLOB and CLOB - Binary Large Object and Character Large Object. A column can be declared to hold one of these types.

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

3. Typically large objects are stored separately, with the database proper containing a reference to where the object is stored, to avoid needing to read the whole object every time a row in the table is accessed.

B. There are times, though, when the objects stored in a table do have a structure, but the structure changes frequently, or varies from object to object, or individual attributes themselves have internal structure (they are not atomic).

1. This kind of data can be stored in certain so-called NoSQL database systems, which we will consider further at the end of the course.

2. The reading for today in the book discussed some semi-structured approaches to storing data such as JSON, or XML.

   a. JSON, XML or similar types of data can be stored in a relational database as a CLOB or BLOB, but application code written in a traditional programming language must be used to get at the internal structure.

   b. Some NoSQL database systems do include facilities for accessing the internal structure of data represented this way, but that's not something we'll talk about.

   c. A major driving force behind movement to NoSQL for certain applications is related to performance and specifically to the performance implications of support for ACID transactions. We will get back to this when we look at NoSQL databases later in the course.

C. For now, we want to focus on ways the relational model can be adapted to more seamlessly support working with more traditional OO applications.

III.Differences Between the OO and Relational Models

A. In regard to structured data, both the OO and relational models have some similarity.

1. OO is based on classes, and all objects of the same class have the same fields with the same data types.

2. The relational model is based on tables, and all rows in the same table have the same columns with the same data types.

   However, before we can talk further about relational support for OO, we need to consider how the two models actually differ in significant ways.

B. The OO and relational models have very different histories.

1. OO: Comes out of the world of discrete simulation. Much subsequent work was motivated by the development of GUI's in the desktop/laptop/mobile device 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).

2. 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, and most data is of atomic types (numbers, strings, dates etc.)

C. The OO and relational have very different fundamental concepts.

1. 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 doe 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.

2. 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 importantly, 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 <u>value</u> of certain of its attributes - something which is semantically meaningful, and doesn’t change from run to run of the same program.

3. 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;
   ...
```

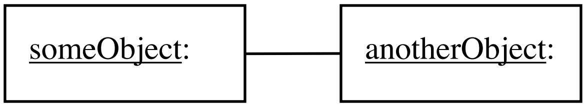
strictly speaking, this is not necessary in an OO system (unless some sort of map is going to be needed to locate the objects, of course) However, something like this is often needed in creating a relational scheme if one cannot guarantee that the remaining attributes of an object will be unique.

4. That is, we have the following:

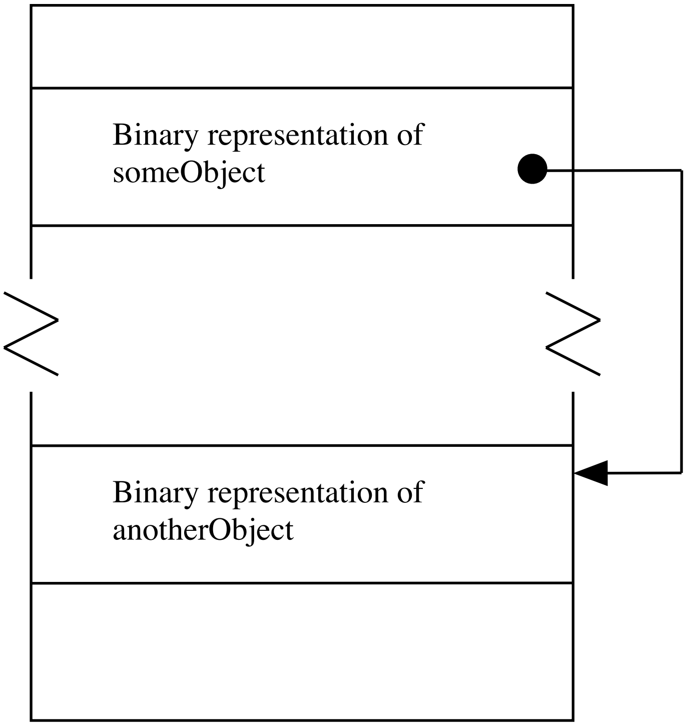
OO:	identity ≠ state
relational:	identity is the same thing as state

D. 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.

1. In an OO system, relationships (associations) are handled by <u>pointers</u> (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



PROJECT

2. In a relational system, relationships are managed by <u>foreign keys</u>. Often, relationships are modeled by tables as well - i.e. a relational database table may correspond to an entity (object) or a relationship (association between objects).

3. There are two consequences of this difference:

   a. Foreign keys are semantically 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 if asked?

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

   b. But 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 search or index lookup. Quantitatively, the time difference - even in the presence of an index - may take much less than a microsecond versus a few milliseconds if a disk access is needed to resolve the reference - a ratio of over 1000:1.

c. 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.

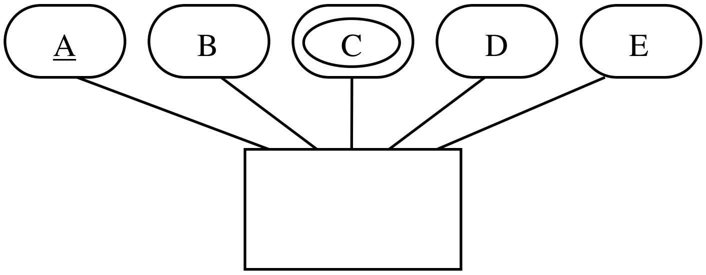
OTOH, as I was revising a similar lecture several years ago, someone asked me "where do I find room 209?" After playing "20 questions" for a while, I determined that what she was looking for was the secretary's office - at which point giving directions became easy. Like most people, I can remember where 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) modeled by pointer
relational: relationship modeled by foreign key

E. Normalization requirements result in often needing to store individual objects in multiple tables.

1. 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



PROJECT

What does this normalize to for 4NF?

ASK

Two tables: (ABDE) and (AC) to satisfy the requirements of 4NF

2. However, OO classes can contain attributes which are collections of various sorts, and can contain non-atomic attributes.

3. This has some profound implications, which someone has illustrated as follows:

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

   ASK$_0$ 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 difference between OO and relational representations for information is summed up neatly by the following diagram from Sadalage:

   PROJECT Sadalage Figure 1.1

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

1. 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.

2. 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.

3. In a generic database, one often needs to use the data in ways that might not have been anticipated when the database was designed. This can be easily done in SQL, of course, but it means breaking encapsulation if he data being stored actually represents objects that belong to OO classes.

   What does one do then? Wait for new methods to be coded? Break encapsulation? Neither seems like a good solution.

G. Differences like these between the two models create challenges for application programmers who are writing an OO program but need to store data in a relational database. This, in turn, engenders greater development effort for applications, and flies in the face of approaches such as agile development.

IV. Approaches to Dealing With These Issues

A. One possibility is to develop a new object-oriented database model. There was a lot of interest in this at one time, but efforts to produce a standardized OO model have been unsuccessful and there is currently little work in this regard except for certain niche areas.

1. For example, the 3rd and 4th editions of our text had a separate chapter on “Object-Oriented Databases”, while the 5th and 6th editions combine this with the Object-relational approach to produce a single chapter on “Object-Based Databases”, and the edition we are using includes material on OO support features in a chapter on Complex Data Types.

2. 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. Another possibility is to extend the relational model to make it a better match for the requirements of OO - creating what is often known as an object-relational model.

1. Several of the extensions to SQL in SQL 1999 (and beyond) have been concerned with reducing the impedance mismatch between the relational model and the needs of OO systems, by moving away from the restrictions the straight relational model imposes on attributes.

a. Allowing non-atomic data types

(1)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.

(2)Columns which may store a collection (e.g. set, array) of value 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 standard 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 attribute for each row in the table which is itself another table. The DBMS transparently translates between a collection field in an object and a table columns in a database row.

(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 that can be looked up quickly in a hash index) rather than a key that must be looked up by searching or in a more sophisticated index. Again, efficiency is a key motivation.

b. Direct support for inheritance

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

(2)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)

(3)Note that these are really two different extensions - in the one case the domains of individual columns use inheritance; in the other case, whole table structures use inheritance. A given database implementation may, of course, implement neither, either or both.

c.Storing procedures in the database along with the data

(1)As methods of specific data types.

(2)As "stand-alone" functions or procedures.

(3)This is actually what something like `sqlj` does - SQL code embedded in a Java program is compiled into stored procedures stored in the database by the `bind` operation.

d. Extensions intended to make SQL 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.

2. Various commercial systems developed during the 1990's incorporated some or all of these facilities, which became part of the 1999 and subsequent SQL standards - 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, since incorporation of these ideas into commercial systems often preceded standardization.)

C. Finally, one might simply live with the differences. Essentially, this is what many existing systems do. 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.

1. Example: This is essentially what you are doing in your programming project by embedding SQL statements in Java code.

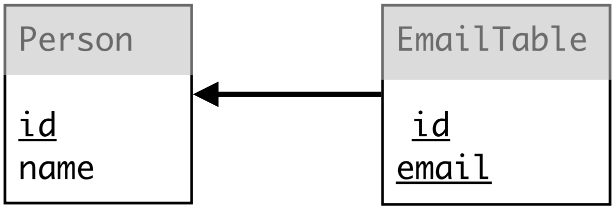
2. To do this, one must map certain OO constructs to corresponding relational constructs.

a) If a class contains a field that is actually a collection, the collection might become a separate table in the relational database. Each row will contain a value of the collection element plus the primary key of the object in which it is contained.

Example:

```
class Person {
    private int id;
    private String name;
    private Set <String> emails;
}
```

might map to:



PROJECT

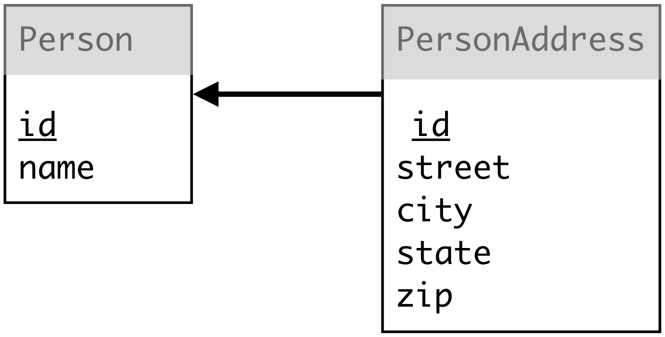
b) If a class contains a field that is itself an object of another class, then objects of that class might be stored in a separate table with the values of their own fields plus the primary key of the table to which it belongs.

Example:

```
class Person {
    private int id;
    private String name;
    private Address address;
}

class Address {
    private String street;
    private String city;
    private String state;
    private int zip;
}
```

might map to:



PROJECT

c) An inheritance hierarchy might be mapped to separate tables for the base class and the subclasses - with the base class table containing the inherited attributes and the subclass tables each containing the subclass attributes plus the primary key of the superclass object.

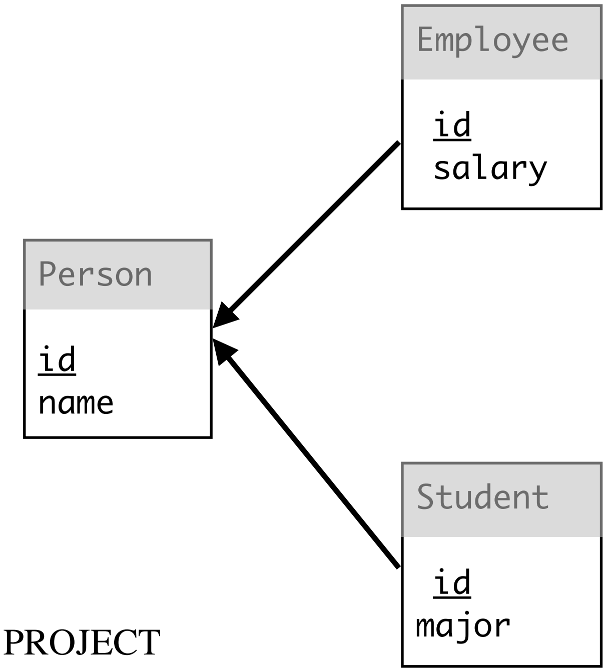
Example:

```
class Person {
    private int id;
    private String name;
}

class Employee extends Person {
    private int salary;
}

class Student extends Person {
    private String major;
}
```

might map to:



PROJECT

3. Actually, mapping like this is how object-relational features might actually be implemented - but transparently - in a DBMS that supports such features.