Objects in DBMSs: A Bit of History

- In the 1990s, object-oriented languages grew in popularity.
  - led people to attempt to add OO features to DBMSs
- One motivation was the desire to support complex data types:
  - multivalued attributes
  - attributes with structure
    - example: an address
      - want to keep its components separate
      - also want to be able to treat it as a unit
  - abstract data types (ADTs) with their own operators/methods
    - including types for nontraditional data: sounds, images, etc.
- Sought to address an impedance mismatch between:
  - the way that applications represent data in memory
  - the way that data can be stored using the relational model
Objects in DBMSs: A Bit of History (cont.)

• One approach: add persistence to an object-oriented language
  • allow objects to persist on disk
  • extend the OO language to support queries, etc.
  • use the language to create an object-oriented DBMS
  • versions of this still exist (e.g., JDO: Java Database Objects)

• At the time, many people thought that object-oriented DBMSs would replace RDBMSs, but they didn’t.

• Instead, object-oriented features have been added to relational systems to create object-relational DBMSs.

Object-Oriented Data Models

• We’ll look briefly at two different models:
  1. ODL (object definition language)
  2. the object-relational model

• In ODL, relations are not central to the model, although we can express a relation as a set of objects.

• In the object-relational model, relations are still central.
ODL (Object Definition Language)

- a "pure" object-oriented data model
- It was originally intended as the data-definition portion of a query language for OO DBMSs.
  - like CREATE TABLE statements in SQL
- More often, it is used when designing a database.
  - like an ER model
  - even if the resulting database is not object-oriented
- It will allow us to discuss some of the OO features that can be incorporated into DBMSs.

ODL Classes

- Use a separate class for each type of entity.
- Classes have three types of properties:
  - attributes: describe an entity
    - can have structure (unlike relational attributes)
  - relationships: connections between an object of a class and one or more other objects
  - methods that can be applied to objects of the class

```java
class Course {
    attribute string name;
    attribute string semester;
    relationship Room location;
    relationship Set<Student> enrolledStudents;
    Set<string> studentNames();  // method declaration
}
```

- ODL also allows for inheritance: classes can have subclasses.
ODL Data Types

• Basic types:
  • atomic types:
    • integer, float, character, string, boolean, etc.
  • class types

• Structured types, which combine basic types:
  • structures:
    Struct name {
      type1 fieldname1,
      type2 fieldname2,
      ...
    }
  • example:
    Struct address {
      string street,
      string city,
      ...
    }
  • collection types:
    Array<T>, Bag<T>, List<T>, Set<T>, ...

• Collection types allow us to have multi-valued attributes.

Attributes vs. Relationships

• An attribute describes an object.
  • the value of the attribute “belongs to” the object
  • the value cannot be accessed except by means of the object

• A relationship connects an object to one or more independent objects.
  • can be thought of as a set of one or more references to objects that are stored elsewhere

• We don’t typically use class types for attributes.
  • implies that we’re embedding one object inside another one
  • instead, use struct types for attributes with structure
  • an attribute with a class type should really be a relationship!
Capturing Relationships in ODL

• Recall: a relationship set is often translated into a relation.
  • example: we would need three relations for the following

  Course(id, name, numCredits, ...);
  Student(id, name, address, ...);
  Enrolled(student_id, course_id);

• In ODL, we only need two classes: Course and Student.
  • capture Enrolled within the class for one or both entity sets:

    class Course {
      ... relationship Set<Student> enrolledStudents;
    }

Capturing Relationships in ODL (cont.)

• Recall: a relationship set is sometimes captured within the relation for one of the connected entity sets:

  Course(id, name, ..., room);
  Room(id, building, roomNumber, ...)

  • this works for what types of relationships?

• In ODL, we can also capture many-to-many relationships in the classes of the connected entity sets.
  • example: the enrolledStudents relationship

• However, we need a separate "relationship" class for ternary and higher-degree relationships.
Data Types of ODL Relationships

- Two options, depending on the cardinality constraints:
  - a class type:
    • example: a course meets in at most one room
      ```java
class Course {
    relation Room location; 
  }
```
  - a collection type applied to a class type:
    • example: a course can have more than one student
      ```java
        class Course {
          relation Set<Student> enrolledStudents;
      }
      ```

Inverse Relationships

- In ODL, we often capture a binary relationship set by putting a relationship property in both of the entity sets' classes:

  ```java
  class Course {
    relation Set<Student> enrolledStudents;
  }
  class Student {
    relation Set<Course> courses;
  }
  ```

- Two relationships that represent the same relationship set are known as inverse relationships.
Inverse Relationships (cont.)

- If relationship R connects an object A with objects B₁, …, Bₙ, then the inverse relationship of R connects each of the objects B₁, …, Bₙ with object A.
  - ex: for a Student s and a Course c, if s is in c.enrolledStudents, then c should be in s.courses

- Indicate this in ODL as follows:

  ```
  class Course {
    relationship Set<Student> enrolledStudents
    inverse Student::courses;
  }
  class Student {
    relationship Set<Course> courses
    inverse Course::enrolledStudents;
  }
  ```

Object Identifiers (OIDs) and Primary Keys

- In an OO DBMS, every object is given a unique identifier (OID).
  - allows the DBMS to implement relationships
  - the actual OID values are not visible to users and cannot be queried

- Because of OID, primary keys are optional in ODL.

- However, they can still be specified:

  ```
  class Course (key (name, semester)) {
    attribute string name;
    attribute string semester;
    attribute int numCredits;
    attribute int cost;
    …
  }
  ```
Object-Relational Model

• Extends the relational model by adding OO features.
• Tuples are treated like objects.
• The added features include:
  • structured types for attributes and relations, including:
    • user-defined types
    • type inheritance
  • collection types – including sets
  • reference types
  • methods

Reference Types

• The object-relational model allows an attribute to store a reference (or a collection of references) to other tuples.
• Here’s a schema for Course that uses reference types:

  Course(name, semester, numCredits, location(*Room), enrolledStudents(*Student))

  • the * indicates a reference
  • location is a reference to a single Room tuple
  • enrolledStudents is a set of references to Student tuples (as indicated by the braces)

• Attributes with reference types are like relationship fields in ODL.
The Object-Relational Model and SQL

- Support for many – but not all – of the features of the object-relational model have been added to SQL.
- The SQL-99 standard includes some object-relational features.
- The SQL-2003 standard adds support for additional ones.
- RDBMs differ in the degree to which they support these features.

Working with Structured and Reference Types

- Assume this object-relational schema:
  
  ```
  Student(id, name, address(street, city, state, zip), ...)
  Room(id, name, capacity)
  Course(name, semester, numCredits, location(*Room), enrolledStudents(*Student))
  ```

- To access a field in an attribute with a structured type, we use the . operator:
  ```
  SELECT S.name, S.address.city
  FROM Student AS S;
  ```

- To access an attribute in a referenced tuple, we use the - > operator:
  ```
  SELECT C.name, C.location->roomNumber
  FROM Course AS C;
  ```
References and Queries

- Find the room numbers of all courses that meet in the Sever Hall:
  \[
  \text{SELECT } \text{C.location->roomNumber} \\
  \text{FROM Course AS C} \\
  \text{WHERE C.location->building LIKE "Sever\%";}
  \]

- Note that we're accessing information from tuples in the Classrooms relation without needing to join Courses and Classrooms.

Array Types

- We can specify an attribute that is a fixed-sized array:
  \[
  \text{CREATE TABLE Student (} \\
  \quad \text{name VARCHAR(30),} \\
  \quad \quad \text{phones CHAR(10) ARRAY[5]} \\
  \text{);} 
  \]

- There are special operators to:
  - access an array element:
    \[
    \text{phones[2]}
    \]
  - get the number of elements in the array:
    \[
    \text{CARDINALITY(phones)}
    \]
Multiset Types

• A multiset is an unordered collection in which a given element can appear more than once.
  • also known as a bag

• Support for multiset types was added in SQL-2003. Example:

```sql
CREATE TABLE Student (
  name VARCHAR(30),
  ...,
  phones CHAR(10) ARRAY[5],
  courses REF(Course) MULTISET
);
```

• Differences between array types and multiset types:
  • elements of an array have an associated position; elements of a multiset do not
  • an array has a fixed size; a multiset does not

Multiset Types (cont.)

• With multisets, we no longer need a separate relation to capture many-to-many relationship sets.

```sql
CREATE TABLE Student (
  name VARCHAR(30),
  ...,
  phones CHAR(10) ARRAY[5],
  courses REF(Course) MULTISET
);
```

```sql
CREATE TABLE Course (
  name VARCHAR(30),
  semester CHAR(6),
  location REF(Room), ...
  students REF(Student) MULTISET
);
```
ER Diagram → Object-Relational Schema

• Given the extensions to SQL, we can now capture additional aspects of ER models.

• Example from earlier in the semester:

\[ \text{course} \]

\[ \text{exam dates} \]

\[ \text{start time} \]

\[ \text{end time} \]

\[ \text{length} = \text{end time} - \text{start time} \]

• how would we capture exam dates?

• how would we capture length?

Queries Involving Complex-Type Values

• The SQL standards specify a number of special operators that make it easier to work with complex-type values in queries.

• For example:

```
SELECT S.name
FROM Student AS S
WHERE '123-456-7890' MEMBER OF S.phones;
```

where S.phones is a multiset.

• Another example:

```
SELECT S.name, P.phone
FROM Student AS S, unnest(S.phones) AS P(phone);
```

would create tuples of the form (student-name, student-phone), with a separate tuple for each of the student's phone numbers.
Queries Involving Complex-Type Values (cont.)

```
SELECT S.name, P.phone
FROM Student AS S, unnest(S.phones) AS P(phone);
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>phones</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>(456-5678, 666-7890, 234-1234)</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>(777-7777, 111-1111)</td>
</tr>
</tbody>
</table>

Implementation Issues

- In pure OODBMSs, transferring objects to and from disk involves translating references/pointers:
  
  memory addresses $\leftrightarrow$ addresses stored on disk

  This process is known as pointer swizzling.

- Other issues arise in both OO and object-relational DBMSs, such as:
  - attributes that have collection types can have arbitrarily large sizes. how do you store them efficiently?
  - what if the methods associated with user-defined types have bugs?
Implementing Array and Multiset Types

- Option 1: use our standard approach for variable-length records, and store the array or multiset within the record.

0 4 8 12 16 23 31
16 23 31 75 1234567 foobar xx

- can just store an offset to the first element, and use simple arithmetic to access the other elements

- Option 2: store the array or multiset on a separate page, and store a reference to it in the record for the tuple

0 4 8 12 16 23 31
16 23 31 39 1234567 foobar xx 25, 2

Implementing Array and Multiset Types (cont.)

- Option 3: use separate tables for array or multiset attributes.

- example:

<table>
<thead>
<tr>
<th>student_id</th>
<th>name</th>
<th>...</th>
<th>phones</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>...</td>
<td>123-456-7890, 444-777-0000</td>
</tr>
<tr>
<td>77777777</td>
<td>Ted Codd</td>
<td>...</td>
<td>333-888-2222, 111-666-4444</td>
</tr>
<tr>
<td>12121212</td>
<td>Alan Turing</td>
<td>...</td>
<td>911-101-1100, 411-101-2222</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_id</th>
<th>name</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>...</td>
</tr>
<tr>
<td>77777777</td>
<td>Ted Codd</td>
<td>...</td>
</tr>
<tr>
<td>12121212</td>
<td>Alan Turing</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_id</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>123-456-7890</td>
</tr>
<tr>
<td>12345678</td>
<td>444-777-0000</td>
</tr>
<tr>
<td>77777777</td>
<td>333-888-2222</td>
</tr>
</tbody>
</table>

... ... ... ...