Complex Data;
Objects in Database Systems

Computer Science 460/660
Boston University
Fall 2013
David G. Sullivan, Ph.D.

Course Checkpoint

• Recall our earlier discussion of the two layers of a DBMS:
  • the logical layer
  • the storage layer or storage engine

• We've spent the last few weeks looking at how a transactional storage engine is implemented.

• Earlier in the course, we studied two logical models:
  • entity-relationship
  • relational

• We'll now begin looking at other logical models.
Objects in DBMSs: A Bit of History

• In the 1990s, object-oriented languages grew in popularity.

• This led people to attempt to add key aspects of these languages 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
  • types that inherit from other types
  • abstract data types (ADTs) with their own operators/methods
    • including types for nontraditional data like sounds, images, etc.

Objects in DBMSs: A Bit of History (cont.)

• One way: 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

```cpp
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:
  - collection types:
    - `Array<T>`, `Bag<T>`, `List<T>`, `Set<T>`, ...
  - structures:
    - `Struct` `name` `{` `type1 fieldname1`, `type2 fieldname2`, `...` `}
    - example:
      - `Struct address {string street, string city, ...}`
**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

```plaintext
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:

```plaintext
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
      ```
      class Course {
          ... 
          relationship Room location;
          ...
      }
      ```
  - a collection type applied to a class type:
    - example: a course can have more than one student
      ```
      class Course {
          ... 
          relationship 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 {
    ...,
    relationship Set<Student> enrolledStudents;
}
class Student {
    ...,
    relationship 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:

```java
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 OIDs, primary keys are optional in ODL.

- However, they can still be specified:

```java
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
  - tuple identifiers: like object identifiers in ODL
  - reference types
  - methods
Nested Relations

- Because the object-relational model includes collection types, we can have an attribute that is a set – i.e., a relation.
  - thus, we end up with nested relations
  - example:

    Courses(name, semester, numCredits, location, enrolledStudents(name, yearOfGrad, ...));

<table>
<thead>
<tr>
<th>name</th>
<th>semester</th>
<th>...</th>
<th>enrolledStudents</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscie119</td>
<td>fall</td>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Turing, Alan</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sullivan, Perry</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cscie268</td>
<td>spring</td>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Codd, Ted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sullivan, Perry</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reference Types

- Our schema for Courses on the previous page is not ideal.
  - why?

- To address these problems, the object-relational model allows an attribute to store a reference (or a collection of references) to tuples from other relations.
Reference Types (cont.)

• Here’s an improved schema for Courses that uses reference types:

```plaintext
Courses(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 relationships 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.
  • a notable exception is that it does not support nested relations

• The SQL-2003 standard adds support for nested relations.
User-Defined Types (UDTs)

- User-defined types are similar to classes in ODL.
  - can include attributes and methods
  - however, they don’t have separate relationship properties
  - in addition, they can’t include primary keys
    - instead, the keys are part of the definition of the relation

- Syntax: `CREATE TYPE typename AS (attributes) methods;`

- Example:
  ```sql
  CREATE TYPE AddressType AS (street VARCHAR(30), city VARCHAR(20), zip CHAR(5))
  METHOD getState() RETURNS CHAR(2);
  ```

- As in ODL, methods are declared in the type definition, but they are defined separately.
  - we won’t cover method definitions in this course

User-Defined Types (UDTs) (cont.)

- We can use a UDT as the type of:
  - an attribute:
    ```sql
    CREATE TYPE StudentType AS (name VARCHAR(30),
    address AddressType, ...
    );
    ```
  - we say that such attributes have a structured type or row type

- a relation:
  ```sql
  CREATE TYPE CourseType AS (name VARCHAR(30), semester CHAR(6), ...
  );
  CREATE TABLE Courses OF CourseType;
  ```
User-Defined Types (UDTs) (cont.)

- If a UDT is used as the type of a relation, the primary key is specified in the `CREATE TABLE` statement as follows:

  ```sql
  CREATE TABLE Courses OF CourseType (  
      PRIMARY KEY(name, semester)  
  );
  ```

References

- We can define an attribute to be a reference to a tuple from a relation of a given type.

  ```sql
  CREATE TYPE RoomType AS (  
      building VARCHAR(30), roomNumber CHAR(6));
  CREATE TYPE CourseType AS (  
      name VARCHAR(30), semester CHAR(6), ...  
      location REF(RoomType), ...  
  );
  ```

- We can also specify that the referenced tuple must come from a particular relation of that type:

  ```sql
  CREATE TABLE Classrooms OF RoomType;
  CREATE TABLE Courses OF CourseType (  
      location REF(RoomType)  
      SCOPE Classrooms);  
  ```
Reference Columns

- For tuple identifiers, SQL uses values from a special reference column.

- The values stored in this column can be:
  - system-generated
  - user-generated
  - derived from the primary key

- We'll limit ourselves to system-generated reference columns, which are specified when creating the table:

  ```sql
  CREATE TABLE Students OF StudentType (REF IS studentID SYSTEM GENERATED);
  ```

- Unlike OIDs in ODL, tuple identifiers are visible to users – just like any other attribute.

Accessing Fields in Structured and Reference Types

- To access a field in an attribute with a structured type, we use the . operator:

  ```sql
  CREATE TYPE StudentType AS (name VARCHAR(30), address AddressType, ...);
  CREATE TABLE Students OF StudentType;
  SELECT S.name, S.address.city FROM Students AS S;
  ```

- To access an attribute in a referenced tuple, we use the -> operator:

  ```sql
  CREATE TYPE CourseType AS (... location REF(RoomType), ...);
  CREATE TABLE Courses OF CourseType ...;
  SELECT C.name, C.location->roomNumber FROM Courses AS C;
  ```
References and Queries

• Find the room numbers of all courses that meet in the Sever Hall:

```
SELECT C.location->'roomNumber'
FROM Courses AS C
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:

```
CREATE TYPE StudentType AS {
  name VARCHAR(30),
  address AddressType,
  phones CHAR(10) ARRAY[5]
};
```

• There are special operators to:
  • access an array element:
    ```
    phones[2]
    ```
  • get the number of elements in the array:
    ```
    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 TYPE StudentType AS (  
    name VARCHAR(30),  
    address AddressType,  
    phones CHAR(10) ARRAY[5],  
    courses REF(CourseType) 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

Nested Relations in SQL-99

- Nested relations are not really part of SQL-99, because that standard doesn't support sets.

- Instead, we still need to use a separate relation to capture many-to-many relationship sets.

- Example:
  ```sql
  CREATE TABLE Students OF StudentType (  
    REF IS studentID SYSTEM GENERATED  
  );
  CREATE TABLE Courses OF CourseType (  
    REF IS courseID SYSTEM GENERATED  
  );
  CREATE TABLE EnrolledIn (  
    student REF(StudentType) SCOPE Students,  
    course REF(CourseType) SCOPE Courses  
  );
  ```
Nested Relations in SQL-2003

- An attribute with a multiset type is a nested relation.
- Thus, we no longer need a separate relation to capture many-to-many relationship sets.

```sql
CREATE TYPE StudentType AS (
    name VARCHAR(30),
    address AddressType,
    phones CHAR(10) ARRAY[5],
    courses REF(CourseType) MULTISET
);

CREATE TYPE CourseType AS (
    name VARCHAR(30),
    semester CHAR(6),
    location REF(RoomType), ...
    students REF(StudentType) 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:

```
course

exam dates

...               ...               ...

start time            end time

length

=end time – start time
```

- how would we capture exam dates?
- how would we capture length?
Specifying Complex-Type Values

- When inserting tuples in a relation, how do we specify array, multiset, and structured values?

- We can specify an array or multiset as follows:
  
  ```
  ARRAY["123-456-7890", "777-666-5555", ...]
  MULTI SET["45678", "12345", "67890", ...]
  ```

- We can specify an attribute that has structure as follows:
  
  ```
  new AddressType("33 Oxford Street", "Cambridge", "MA", "02138");
  ```

  [assuming the following definition of this type:
  ```
  CREATE TYPE AddressType AS {
    street VARCHAR(30),
    city VARCHAR(20),
    state CHAR(2),
    zip CHAR(5));
  ```
  ]

Example of Specifying Complex-Type Values

- Recall our `StudentType` definition:

  ```
  CREATE TYPE StudentType AS {
    name VARCHAR(30),
    address AddressType,
    phones CHAR(10) ARRAY[5],
    courses REF(CourseType) MULTI SET
  };
  ```

- If we have a table `Students` of type `StudentType`, how would we insert a tuple in this relation?

  - here’s everything but the multiset:

    ```
    INSERT INTO Students VALUES (
      "Ted Codd",
      new AddressType("IBM Almaden", "San Jose", "CA", "95120"),
      ARRAY["123-456-7890", "444-555-7777"],
      ...
    );
    ```
Example of Specifying Complex-Type Values (cont.)

- It's tricky to insert the multiset in this case because the values in the reference column are system-generated:

  ```sql
  CREATE TABLE Courses OF CourseType {
    REF IS courseID SYSTEM GENERATED
  };
  ```

- Solution: use a nested query:

  ```sql
  INSERT INTO Students VALUES (  
    "Ted Codd",
    new AddressType("IBM Almaden", "San Jose", "CA", "95120"),
    ARRAY["123-456-7890", "444-555-7777"],
    MULTiset(SELECT courseID FROM Course
            WHERE name = "CS165"
            OR name = "CS182");
  );
  ```

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:

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

  where S.phones is a multiset.

- Another example:

  ```sql
  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.
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.

- Option 2: store the array or multiset on a separate page, and store a reference to it in the record for the tuple
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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</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>
<tr>
<td>...</td>
<td>...</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>77777777</td>
<td>444-777-0000</td>
</tr>
<tr>
<td>12121212</td>
<td>333-888-2222</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>