Object-Oriented and Object-Relational Data Models

Object-Oriented Data Model

Overview

ODL — **O**bject **D**efinition **L**anguage.

Basic notions:

- Classes;
- Attributes;
- Relationships;
- Methods;
- Exceptions;

C/C++ -style syntax.

Complex type system.

ODL Syntax

Classes.

```
class <name> {
              <list of properties>
             }
```

Referencing a property:

```
<class>::<property>
```

Attributes.

Simplest of "properties".

```
attribute <type description> <name>
```

Type descriptions can be simple (`string, integer, enum`) of complex (`Struct, List, Set`).

Relationships.

Another type of "properties". Established between different classes.

```
relationship <class description> <name>
```

Relationships are treated as *references*. `Name` is the name of the attribute for storing the reference(s). `Class description` specifies the type of the reference collection (single reference, set, list, etc...) and the name of the class to the objects of which the references would point.

Inverse Relationships.

```
relationship <class description> <name>
             inverse <class>::<relationship name>
```

Example 1 *Descriptions of classes for the Library database.*

```
class Book {
      attribute string ID;
      attribute integer ISBN;
      attribute string Title;
      relationship Set<Author> Authors
                   inverse  Author::wrote;
      relationship PubCompany publisher
                   inverse PubCompany::Books;
      attribute integer year;
           }

class Author {
       attribute string Name;
       attribute string Country;
       relationship Set<Book> wrote
            inverse Book::Authors;
            }

class PubCompany {
        attribute string Name;
        attribute Struct
             { string street,
               string city,
               integer zip,
               string country,
             } Headquaters;
         relationship Set<Book> Books
                inverse Book::publisher;
      }
```

Types of relationships.

Class A → Class B	Class B → Class A	Type of relationship
$A \rightarrow B$	$B \rightarrow A$	one-to-one
$A \rightarrow \texttt{Set} < B >$	$B \rightarrow A$	one-to-many
$A \rightarrow B$	$B \rightarrow \texttt{Set} < A >$	many-to-one
$A \rightarrow \texttt{Set} < B >$	$B \rightarrow \texttt{Set} < A >$	many-to-many

Methods

Third type of "properties".

Signatures of methods, similar to C/C++ declarations.

```
<type> <Name> (<parameters>) [ raises (<Exceptions>)]
```

- Parameters: in, out, inout.
- Exceptions:

Types

- *Atomic (simple) types.* `integer`, `float`, `char`,`string` `boolean`, `enum`.
- *Class Names.* E.g., `Book` is a type.
- *Structured Types*:
  - *Set*;
  - *Bag* or multiset;
  - *List* (position important);
  - *Array*: `Array<Type,i>`.
  - *Dictionary*: `Dictionary<Type1,Type2>`. Pairs of values are stored. `Type1` - *key type*, `Type2` - *range type*.
  - *Structures*.

Types can be embeddd into one another.

```
Array < List <Books>, 25>
Dictionary < Struct Addr (string street, string city, integer zip),
             Author >
```

Types of Relationships:

- Class type;
- Single (use of a) collection type (set, list, bag, array, dictionary).

Multiway Relationships

Not supported.

Trick. Suppose R is a relationship between classes $C_1, \ldots C_m$. Define a new class C and include many-to-one relationships between C and each of C_is in it.

This also allows to introduce descrptive attributes.

Subclasses and Inheritance

```
class <Name> extends <ClassName> { <extra properties>}
```

Multiple inheritance.

```
class <Name> extends <ClassName1>: <ClassName2>
       { <extra properties>}
```

Attribute inheritance may be undefined.

Extents

A extent of a class is the actual collection of objects with the type specified by the class.

Extents have different names from classes and can be specified in class declaration.

```
class Book (extent Library) {
          ...
      }
```

Keys

ODL objects have unique identity (OIDs).

Keys are not necessary but may be convenient.

Keys are associated with extents !

Declaration of 1 key:

```
class <ClassName>
       ( extent  <ExtentName>  key ( <AttributeNames>))
       {
         ...
       }
```

(Obviously, all attributes declared as part of a key must be defined in the class).

Declaration of multiple keys:

```
class <ClassName>
       ( extent  <ExtentName>  key <AttributeNames> )
       {
         ...
       }
```

For example.

```
class Book
       ( extent  Books  key ISBN, LibCode )
       {
         ...
       }
```

declares **two different keys**, while

```
class Book
       ( extent  Books  key (ISBN, LibCode) )
       {
         ...
       }
```

declares one key with two attributes.

Design Principles

Same as in E-R model.

1. Faithfulness. Follow the specifications.
2. Redundancy Avoidance. Unless necessary of explicitly desired, avoid including redundant elements in the model.
3. Simplcity. Out of two competing designs, the simpler one is typically better.
4. Relationship Selection.

Object-Relational Models

Compare and Constrast

Relational Databases	Object Databases
• Simple Structure	• Complex Structure
• Easy-to-query (SQL)	• ??
• Modeling can be hard	• Models complex domains naturally
• Commercially successful	• Emerging

The Best of Both Worlds

Enter The Object-Relational Model.

What *is* Object-Relational Model ?

Relational Model+

- Structured Types for attributes;
- Methods;
- Tuple Identity;
- References.

Nested Relations

Allowing relational attributes have strucured types leads to *nested* relations.

- A atomic type (integer, real, boolean, character, string) can be a type of an attribute in a relation.
- Let $T_1, \ldots T_M$ be attribute types and $A_1, \ldots, A_M$ be attribute names. Let B be another attribute name. Then $B : (A_1 : T_1, \ldots, A_M : T_M)$ is a valid object-relational attribute.
- Let $B_1, \ldots B_N$ be valid object-relational attributes of types (possibly complex) $T_1, \ldots, T_N$. Then $(B_1 : T_1, \ldots B_N : T_N)$ is a valid object-relational schema.

References

Let T be a valid object-relational attribute type. Then $(*T)$ is a reference type to objects of type T and $\{*T\}$ is a reference type to sets of objects of type T.

Methods

Any function associated with a particular object type (object-relational schema) can be registered as a method and used in queries.

Tuple Identity

Each tuple gets a distinct OID. Tuples with same attributes no longer subject to duplicate elimination.

Query Languages ???

Object-relational model is more complex. How do we **query** object-relational data ?

- **OQL (Object Query Language.** OQL is a pure object-oriented query language, but can be used for object-relational data.
- **SQL** Extension. SQL-99 supports creation of complex types (ADTs), registration of methods, references, tuple identity, and extends query language syntax.

Object-Relational Features of SQL-99

User Defined Types

SQL–99 allows for new types to be defined within a database. They are called User Defined Types or UTDs. There are two important types of UTDs in SQL–99:

- New type of a table;
- New type of a table attribute.

Defining UDTs

SQL–99 provides its own version of syntax, designed to represent ODL definitions.

```
CREATE TYPE <name> AS <declarations>
```

Here, `<name>` is the identifier for the name of the UTD and `<declarations>` is the list of standard attribute and new method declarations.

Attribute Declarations

Attribute declarations are same as in `CREATE TABLE` statement, except that UTDs can be types of attributes.

```
CREATE TYPE interval AS (
    start INTEGER,
    end   INTEGER
    );

CREATE TYPE paper AS
   ( title CHAR(200),
     year  INTEGER,
     pages interval
   );
```

Method declarations

Methods are declared in UTDs as follows:

```
METHOD <name>([<arguments>]) RETURNS <type>
```

Here, `<name>` is the name of the method, `<arguments>` are any arguments that the method takes as input (represented as `<name> <type>` pairs) and `<type>` specifies the output type of the method (and the types of input parameters). Methods are specified outside the parentheses that contain attributes declarations.

```
CREATE TYPE volumeType AS
     (
       VolNum INTEGER,
       Year   INTEGER,
       pages  interval
      )
     METHOD PublicationMonth() RETURNS CHAR(30);
```

Method Definitions

Methods can be defined in SQL–99 using *Persistent Stored Modules*, otherwise known as *PSMs*. PSMs provide syntax for describing user defined functions/procedures associated with a specific database (or database schema). We will not study PSMs in detail. Method declaration:

```
CREATE METHOD <name> RETURNS <type>
FOR <utd>
<body>
```

For example:

```
CREATE METHOD PublicationMonth() RETURNS CHAR(30)
FOR volumeType
BEGIN
IF VolNum = 1
   THEN RETURN ''March''
ELSEIF VolNum =2
   THEN RETURN ''June''
ELSEIF VolNum = 3
   THEN RETURN  ''September''
ELSE RETURN ''December''
END IF;
END;
```

Declaring Relations with UDTs

```
CREATE TABLE <name> OF <udt>
```

For example,

```
CREATE TABLE JVolumes OF volumeType;
```

creates table `JVolumes` which has all the attributes of `volumeType` and has a method `PublicationMonth()` associated with it.

Converting ODL relationships

SQL–99 allows for the use of references in UTDs. This allows for representation of ODL relationships. Reference declarations appear as attribute declarations and have the following syntax

```
<attName> REF(<type>) SCOPE <table>
```

Here, `<attName>` is the name of the reference attribute, `<type>` is the type of the object the reference points to and `<table>` specifies the scope (location) of the object to which the reference points in the database.

```
CREATE TYPE volumeType AS (
       VolNum  INTEGER,
       Year    INTEGER,
       pages   interval,
       journal REF(journalType) SCOPE journals
      )
     METHOD PublicationMonth() RETURNS CHAR(30);
```

References (OIDs) must be generated at the "destination" point, i.e., at the table declarations. This is done using the following statement

```
REF IS <attribute> SYSTEM GENERATED|DERIVED
```

Here, `<attribute>` is the name of the attribute that will serve as the OID for the objects in the table. `SYSTEM GENERATED` means that DBMS has to generate unique ids to ensure reference integrity. `DERIVED` means that the DBMS is to use the primary key of the table to generate OIDs.

```
CREATE TYPE journalType AS (
    ID   INTEGER,
    name CHAR(100),
    publisher CHAR(20),
    subject CHAR(20),
    VolsPerYear INTEGER
     );

CREATE TABLE journals OF journalType (
   REF IS ID DERIVED,
   PRIMARY KEY(ID)
   );
```

`REF` and `REF IS` combination allows one to store a single reference value in an attribute. To account for one-to-many and many-to-many relationships one needs to build relationship tables similar to the ones built for E-R diagrams.

Operations with UDTs

Dereferencing

Let x be of type `REF(T)`. Then

- `x->a` is the value of attribute `a` in the tuple `x` points to.
- `DEREF(x)` is the entire tuple `x` points to.

```
SELECT * FROM volumes
WHERE journal->subject = ''Databases'';

SELECT DEREF(journal) FROM volumes
WHERE year = 1999;
```