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
$A \to B$	$B \to A$	one-to-one
$A \to \mathtt{Set} < B >$	$B \to A$	one-to-many
$A \to B$	$B \to \mathtt{Set} < A >$	many-to-one
$A \to \mathtt{Set} < B >$	$B \to \mathtt{Set} < A >$	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 UDTs. UDTs can then be used as

- Type designations of database tables;
- Type designations of database table attributes.

Defining UDTs

The Oracle version of the SQL type definition is:

```
   CREATE [OR REPLACE] TYPE <name> AS OBJECT
     ( <declarations> );
/
```

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

Note: In sqlplus each `CREATE TYPE` command must end with a / on a separate line (semicolumn is not enough).

Attribute Declarations

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

```
CREATE TYPE interval AS OBJECT (
    lower INTEGER,
    upper   INTEGER
    );
/
```

```
CREATE TYPE paper_type AS OBJECT
   ( title CHAR(200),
     year  INTEGER,
     pages interval
   );
/
```

Collection Types: VARRAY

Oracle allows for a variable array type to represent collection types. A variable array is an array of up to a specified number of components.

`VARRAY` can be used as the type of a table attribute.

Defining VARRAY types.

```
CREATE [OR REPLACE] TYPE <name> AS VARRAY(<n>) OF <datatype>
```

For example, the following statement defines a variable array type for storing multiple papers:

```
CREATE TYPE paper_list_type AS VARRAY(20) OF paper_type
```

Inheritence

```
CREATE [OR REPLACE] TYPE <Name> UNDER <TypeName> (
 ...
 );
 /
```

Example:

```
CREATE TYPE Interval_plus_Center UNDER Interval (
   Center Float
  );
/
```

Creating Object-relational tables

Once a type has been created, relational tables of this type can be created as well. The syntax is:

```
CREATE TABLE <name> OF <datatype>
(
 <constraints>
);
/
```

For example,

```
CREATE TABLE Contents OF paper_type;
```

creates table `Contents` with three attributes specified by the `paper_type` type.

Primary keys, foreign keys and other constraints need to be specified:

```
CREATE TABLE Contents OF paper_type
( PRIMARY KEY(title));
```

UDTs can also be used directly as attribute types in "traditional" CREATE TABLE statements.

```
CREATE TABLE tech_report
(
  Report_No varchar(20) primary key,
  report_info paper_type,
  report_date DATE
);
```

Inserting information into Object-relational tables

`INSERT` statement changes to accommodate for insertion of UDT values. An instance of a UDT value is described by the following syntax:

```
<UDT>(<value1>,<value2>,...,<valueX>)
```

where X is the number of fields in the UDT, and the order of values matches the order of fields in the `CREATE TYPE <UDT>` statement.

Similarly, an instance of a `VARRAY` is described using the syntax:

```
<VARRAY_TYPE>(<value1>,...,<valueX>)
```

where X is less than or equal to the size of the variable array.

Example. Inserting a value into the `tech_report` table:

```
INSERT INTO tech_report VALUES('TR-01-08', paper_type('JOXM', 2008, interval(1,20)));
```

Notice that UTD instance descriptions can be nested.

Example. Inserting into a table with a VARRAY attribute.

Conisder the following table:

```
CREATE TABLE Journal_Issue
( Title varchar(50),
  Volume INT,
  No INT,
  YEAR INT,
  articles paper_list_type,
  PRIMARY KEY(Title, Volume, No)
);
```

Here is an `INSERT` statement for this table:

```
INSERT INTO Journal_Issue
VALUES('SIGMOD Record', 10, 1, 2000,
        paper_list_type(paper_type('Databases are Great', 2000, interval(1,10)),
                        paper_type('Report of SIGMOD-1999', 2000, interval(11,15),
                        paper_type('Oracle's Secrets', 2000, interval(16,29))
                        )
        );
```

Accessing the data

Note: Bad news: VARRAY attributes can be accessed only via PL/SQL. You cannot ask "Show me the second article in each issue" from within pure SQL.

Access structured values, however, is available from SQL. SQL uses "." notation to traverse the hierarchy of nested attributes.

```
SELECT t.report_info.title
FROM  tech_report t
WHERE t.report_info.year = 2008;
```

This query outputs the titles of all tech reports written in 2008.

NOTE: in order to access nested attributes you **must include an alias for your table** in the `FROM` clause.