COMP163
Database Management Systems
December 9, 2008
Object and Object-Relational Databases
Chapter 20: Concepts for Object Database
- overview of OO concepts
- mostly familiar to anyone with experience in UML and OOPLs

Chapter 21: Object Database Standards, Languages and Design
- ODMG standard
- translation from EER diagrams to OO schemas

Chapter 22: Object-Relational and Extended Relational Systems
- Object-relational features of SQL-99
- ORDBMS examples: Informix and Oracle
Motivating Question

- Suppose we need to make all data in a Java program persistent.
- What are our options?
Relational Data Model

- Operational definition: 1\textsuperscript{st} Normal Form
- Database consists of relations (tables)
- Relations are composed of tuples (rows)
- Tuples are composed of attributes
- Attributes are constrained by domains
- Domains are primitive (non-structured) data types
Object Persistence Options

- Serialize the objects and store in files
  - easy in languages that support serialization (Java, Python)
  - problem: no transactions or concurrency control
  - limitation: need to load all data into memory to access any data

- Translate object data into relational data and store in an RDBMS
  - translation process is well understood
  - support for transactions and concurrency control
  - problem: impedance mismatch

- Store objects in an object database
Object Data Models

- Object databases replace the relational data model with an object-based or object-oriented data model
- Database consists of class extents (sets of objects)
- Extents are composed of objects
- Objects are composed of attributes
- Attribute values are constrained by data types
- Data types have arbitrary complexity and structure
Users of the data application program(s) interact with the system through OML and OQL processors. The OML/OQL processors communicate with the query and update processor, which in turn interacts with the concurrency and security managers. The index manager is responsible for managing access structures. The ODL processor communicates with the schema (classes), data (objects), and access structures components.

The diagram illustrates the architecture of an OO-DBMS, highlighting the roles of various components and their interactions.
Early OODBMS vendors based their systems on several different object models:

- Versant, ONTOS: persistent C++ objects
- O2: object model based on complex value theory
- Others: persistent Smalltalk objects, Objective-C, etc.

ODMG: The Object Database Management Group

- developed a common model for OODBs
- provides the benefits of standardization in same manner as the standard relational model
- allows for portability of applications and sharing of objects between systems
ODMG Standard Components

- **Object Model**: defines the concepts available for defining an OO schema
  - usual OO things: classes, attributes, methods, inheritance
  - database things: relationships, extents, collections, transactions, DBMS control

- **Languages:**
  - Object Definition Language: ODL
  - Object Query Language: OQL
  - Object Manipulation Language: OML
ODL: Object Definition

- ODL defines the syntax for implementing the object model
  - ODL is the language for defining an object schema

- ODL is actually a family of languages:
  - the ODMG language neutral ODL
  - C++/ODL
  - Java/ODL
  - Smalltalk/ODL

- ODL consists of class declarations
Example Schema: ER

- **DOG**
  - name
  - breed
  - license
  - n

- **OWNS**
  - 1

- **PERSON**
  - name
  - addr
  - phone
CREATE TABLE PERSON
(
    name  VARCHAR(20) NOT NULL,
    addr  VARCHAR(50) NOT NULL,
    phone CHAR(10)    NOT NULL;
    CONSTRAINT PERSON_PK PRIMARY KEY (phone)
);

Since phone is the primary key, it becomes the thing that identifies a person.
CREATE TABLE DOG
( name VARCHAR(20) NOT NULL,
  breed VARCHAR(15) NOT NULL,
  license VARCHAR(10) NOT NULL;
  owner_phone CHAR(10) NOT NULL,
CONSTRAINT DOG_PK PRIMARY KEY(license),
CONSTRAINT DOG_FK FOREIGN KEY
( owner_phone ) REFERENCES PERSON(phone) );

A dog is identified by its license number.

A dog's owner is identified by his/her phone number, since that is a person's primary key.
### PERSON

<table>
<thead>
<tr>
<th>phone</th>
<th>name</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>222-7777</td>
<td>Harsha</td>
<td>22 Lake</td>
</tr>
<tr>
<td>111-2222</td>
<td>Charlie</td>
<td>16 Pine</td>
</tr>
<tr>
<td>333-9999</td>
<td>Regina</td>
<td>801 F</td>
</tr>
</tbody>
</table>

### DOG

<table>
<thead>
<tr>
<th>license</th>
<th>name</th>
<th>breed</th>
<th>owner_phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Snoopy</td>
<td>beagle</td>
<td>111-2222</td>
</tr>
<tr>
<td>004</td>
<td>Ace</td>
<td>shepherd</td>
<td>222-7777</td>
</tr>
<tr>
<td>003</td>
<td>Rover</td>
<td>poodle</td>
<td>222-7777</td>
</tr>
<tr>
<td>012</td>
<td>Spot</td>
<td>mutt</td>
<td>333-9999</td>
</tr>
</tbody>
</table>

Relationships are implemented as foreign keys.
Relational Queries

SELECT P.NAME
FROM PERSON AS P, DOG AS D
WHERE D.NAME = "Snoopy"
    AND D.owner_phone = P.phone;

SELECT D.NAME
FROM PERSON AS P, DOG AS D
WHERE P.NAME = "Harsha"
    AND D.owner_phone = P.phone;

relationships are accessed through joins over the foreign key
class PERSON
  (extent people, key phone)
  {
    attribute string name;
    attribute string phone;
    attribute string addr;
    relationship set<PET> owns
    inverse PET::owner;
  };

extent: set of all instances of the class
key: attribute is unique among all instances (not a primary key)
class DOG
 (extent dogs, key license)
 {
  attribute string name;
  attribute string breed;
  attribute string license;
  relationship PERSON owner
   inverse PERSON::owns;
};

relationships are references to other objects
the inverse relationship implies a consistency constraint
class PERSON : public d_Object {
private:
    d_String name;
    d_String addr;
    d_String phone;
    d_Rel_Set<DOG, "owner"> owns;
};

d_Set<d_Ref<PERSON> > people;

the class extent is a set of references

all classes inherit from d_Object (database object)

one side of the relationship, defined as a set of relationship references
class DOG : public d_Object {
private:
    d_String name;
    d_String breed;
    d_String license;
    d_Ref_Ref<PERSON, "owns"> owner;
};

d_Set<d_Ref<DOG> > dogs;

one side of the relationship, defined as a single relationship reference
extents are sets of object identifiers (OIDs)

extents have persistent names

(notation: circles are names, not pointers or objects)
Object Identifiers

- All database objects are assigned unique object identifiers (OIDs)
  - inherited from d_Object

- An OID gives an object an immutable identity, apart from its value
  - In a relational database, the identity of tuples is determined by their value

- An OID identifies an object regardless of its location in memory, on disk or on network
OIDs Are References

**Dogs**
- **D784**
  - Rover poodle
  - 003
  - P352
- **D996**
  - Snoopy beagle
  - 001
  - P188
- **P352**
  - Harsha
  - 22 Lake
  - 222-7777
  - {D784, D112}
- **D643**
  - Spot mutt
  - 012
  - P080

**Persons**
- **E001**
  - {D784, D643, D996, D112}
- **E009**
  - {P188, P080, P352}
- **P188**
  - Charlie
  - 16 Pine
  - 111-2222
  - {D996}
- **D112**
  - Ace shepherd
  - 004
  - P352

**Miscellaneous**
- **E009**
  - {P188, P080, P352}
- **P080**
  - Regina
  - 801 F
  - 333-9999
  - {D643}
relationships are defined by OID values or sets
Relationships are Reciprocal References

D784
Rover poodle
003
P352

P352
Harsha
22 Lake
222-7777
{D784, D112}

D112
Ace shepherd
004
P352

D996
Snoopy beagle
001
P188

P188
Charlie
16 Pine
111-2222
{D996}

P080
Regina
801 F
333-9999
{D643}

D643
Spot mutt
012
P080
The DBMS is responsible for maintaining the consistency of relationships. Suppose Regina sells her dog to Harsha:

```c
d_Ref<Person> regina = people.select_element("name='Regina'");
d_Ref<Person> harsha = people.select_element("name='Harsha'");
d_Ref<Dog> spot = regina->owns.select_element("name='Spot'");
regina->owns.clear();
harsha->owns.insert_element(&spot);
```
Relationship Maintenance

result of the extent queries
null
harsha->owns.insert_element(&spot);
Relationship Maintenance

The DBMS must correct the inverse relationships
Data Model Spectrum

Where ODBMS Fits

- **Set-based**
  - RDBMS
    - Simple data, simple links
  - ORDBMS
    - Complex data, complex links
- **Nonset-based**
  - Network flat file
    - Simple data
  - ODBMS
    - Complex data

Data Complexity

Query Complexity

Simple

Complex
SQL-99: Object-Relational Support

- Type constructors to specify complex objects
  - UDTs → User Defined Types
  - two kinds of UDTs: rows and arrays
- Mechanism to specify object-identity
- Mechanism for encapsulation of operations
- Mechanism to support inheritance
Type Constructor: Row

- Row types consist of flat domains
- Similar to ER composite attribute
- Example:

```sql
CREATE TYPE Addr_type AS ( 
  street VARCHAR (45),
  city VARCHAR (25),
  zip CHAR (5));
```
Type Constructor: Array

- An array type is specified for attributes whose values are collections
- Similar to ER multi-valued attribute
- Example:

  ```sql
  CREATE TYPE Comp_type AS (  
    comp_name VARCHAR (2),  
    location VARCHAR (20) ARRAY [10]  
  );
  ```
A user-defined type can also be used to specify the row types of a table:

```sql
CREATE TABLE Company OF Comp_type(
    REF IS comp_id SYSTEM GENERATED,
    PRIMARY KEY (comp_name));
```

Syntax to specify object identifiers:

```sql
REF IS <oid_attribute>
    <value_generation_method>
```

Options:
- SYSTEM GENERATED
- or DERIVED
Attributes as References

• A component attribute of one tuple may be a reference:

  CREATE TYPE Employment_type AS (  
      employee REF (Emp_type) SCOPE (Employee),  
      company REF (Comp_type) SCOPE (Company));

  CREATE TABLE Employment OF Employment_type;

• Keyword **SCOPE** specifies the table whose tuples can be referenced by a reference attribute

  * e.company->comp_name
Path Expressions

- Path expressions are used to refer to components of UDTs

- `SELECT E.Employee->Name
  FROM Employment AS E
  WHERE E.Company->Comp_name = 'ABCXYZ';`
Encapsulation of Operations

- A construct similar to the class definition
- Users can create a named user-defined type with its own methods in addition to attributes:

```sql
CREATE TYPE Addr_type AS (  
    street VARCHAR (45),  
    city VARCHAR (25),  
    zip CHAR (5)  
)  
METHOD apt_no ( ) RETURNS CHAR(8);```

User Defined Methods

- Code for methods is supplied externally
  - implemented in some general purpose programming language

```sql
METHOD
CREATE FUNCTION apt_no() RETURNS CHAR(8) FOR Addr_type AS EXTERNAL NAME 'x/y/aptno.class' LANGUAGE 'java';
```
Inheritance in SQL

- **Inheritance** is specified via the **UNDER** keyword

- Example

  ```sql
  CREATE TYPE Manager_type
  UNDER Emp_type
  AS (dept_managed CHAR (20));
  ```

- Manager_type **inherits** all features of Emp_type and has an additional attribute called dept_managed