Object Databases

Chapter 16

Topics

- Motivation
- Conceptual Object Data Model
- Object Extensions in SQL
- Objects in Oracle DBMS
- Current Use of Objects in Databases

Problems with Flat Relations

Consider a relation

Person(SSN, Name, PhoneN, Child)

with:
- FD: SSN \rightarrow Name
- Any person (identified by SSN) can have several phone numbers and children
- Children and phones of a person are not related to each other except through that person

An Instance of Person

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>PhoneN</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-123-4567</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-345-6789</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-123-4567</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-345-6789</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-6543</td>
<td>444-55-6666</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-1111</td>
<td>555-66-7777</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-6543</td>
<td>555-66-7777</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-1111</td>
<td>444-55-6666</td>
</tr>
</tbody>
</table>
Problem 1: Redundancies

- In Chapter 6, it was mentioned that set valued attributes cause redundancy in addition to redundancy due to a relation being not normalized with respect to Functional Dependencies.
- Join Dependency is a constraint used to capture the dependency between set valued attributes.
- Fourth Normal Form:
  - Defined in terms of Join Dependencies
  - We will not study JD or 4th Normal Form

Redundancies in Person

- Due to the FD: \( SSN \rightarrow Name \)
  Joe Public is associated with the SSN 111-22-3333 four times (for each of Joe’s child and phone)! Similarly for Bob Public.
- Due to the JD:
  - \( Person = (SSN, Name, PhoneN) \rightarrowt (SSN, Name, Child) \)
  Every \( PhoneN \) is listed with every \( Child \) SSN.
  Hence Joe Public is twice associated with 222-33-4444 and with 516-123-4567.
  Similarly for Bob Public and other phones/children.

Dealing with Redundancies

- What to do?
- Normalize!
- Split Person according to the JD
- Then each resulting relation using the FD
- We end up with three relations

Normalization removes redundancy

<table>
<thead>
<tr>
<th>Person</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Name</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ChildOf</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhoneN</td>
<td>PhoneN</td>
</tr>
<tr>
<td>SSN</td>
<td>PhoneN</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>516-345-6789</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>516-123-4567</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>212-987-6543</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>212-135-7924</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>444-55-6666</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>555-66-7777</td>
</tr>
</tbody>
</table>
Problem 2: Querying is cumbersome

Get the phone numbers of Joe’s grandchildren

Against the original relation: three cumbersome joins

```
SELECT G.PhoneN
FROM Person P, Person C, Person G
WHERE P.Name = 'Joe Public' AND P.Child = C.SSN AND C.Child = G.SSN
```

Against the decomposed relations is even worse: four joins

```
SELECT N.Phone
FROM ChildOf C, ChildOf G, Person1 P, Phone N
WHERE P.Name = 'Joe Public' AND P.SSN = C.SSN AND C.Child = G.SSN AND G.SSN = N.SSN
```

Objects Allow Simpler Design

Schema:

```
Person(SSN: String, Name: String, PhoneN: [String], Child: [SSN])
```

No need to decompose in order to eliminate redundancy: the set data type takes care of this.

Object 1:

```
( 111-22-3333, "Joe Public", {516-345-6789, 516-123-4567}, {222-33-4444, 333-44-5555} )
```

Object 2:

```
( 222-33-4444, "Bob Public", {212-987-6543, 212-135-7924}, {444-55-6666, 555-66-7777} )
```

Objects Allow Simpler Queries

Schema (slightly changed):

```
Person(SSN: String, Name: String, PhoneN: [String], Child: [Person])
```

- Because the type of Child is the set of Person-objects, it makes sense to continue querying the object attributes in a path expression

**Object-based query:**

```
SELECT P.Child.Child.PhoneN
FROM Person P
WHERE P.Name = 'Joe Public'
```

- Much more natural!

Problem 3: IsA Hierarchy

Person(SSN, Name)
Student(SSN, Major)

Query: Get the names of all computer science majors

Relational formulation:

```
SELECT P.Name
FROM Person P, Student S
WHERE P.SSN = S.SSN and S.Major = 'CS'
```

Object-based formulation:

```
SELECT S.Name
FROM Student S
WHERE S.Major = 'CS'
```

**Student-objects are also Person-objects, so they inherit the attribute Name**
Problem 4: Dealing With Blobs

- The term blob stands for **binary large object**
- Virtually all DBMS support relations to have attributes of type blob
  - E.g., a database with movies may have schema:
    - Movie(Name: String, Director: Person, Video: blob)
  - Where the attribute Video stores the stream of the movie
- The type of operations users want to do on blob types go beyond relational operators
  - E.g., Retrieve frames in the range 20,000 to 50,000
  - But there is no attribute Frame
  - This capability though may be provided via a method associated with blob types

Object Methods in Queries

- For complex objects, operations/methods may be associated with the object type that can be used in queries
- For instance, the method frameRange(from, to) might be a method in class Movie. Then the following query makes sense:

  ```sql
  SELECT M.frameRange(20000, 50000)
  FROM Movie M
  WHERE M.Name = 'The Simpsons'
  ```

Problem 4: The “Impedance” Mismatch

- One cannot write a complete application in SQL, so SQL statements are embedded in a host language, like C or Java
- SQL: Declarative
  - Set-oriented, works with relations, uses high-level operations over them
- Host language:
  - Procedural
  - Record-oriented, does not understand relations and high-level operations on them

Impedance Mismatch

- If program in, e.g., an object oriented programming language needs to access relational data as objects, the following tasks need to be done:
  1. Understand how relational schema should be represented as objects
  2. Translate the object operations to SQL
  3. Map the resultant tuples to objects
- Impedance Mismatch
  - User programs have to keep translating between programming language type system and the relational model because of two different type systems
  - True of not just programs in OO programming language, but also of programs in other languages
Can the Impedance Mismatch be Bridged?

- This was the original idea behind object databases:
  - Use an object-oriented language as a data manipulation language. Since data is stored in objects and the language manipulates objects, there will be no mismatch!
- Problems:
  - Object-oriented languages are procedural – the advantages of a high-level query language, such as SQL, are lost
  - C++, Java, Smalltalk, etc., all have significantly different object modeling capabilities. Which ones should the database use? Can a Java application access data objects created by a C++ application?
  - Instead of one query language we end up with a bunch!
    - One for C++, one for Java, etc.

Is Impedance Mismatch Really a Problem?

- Two main approaches/standards:
  1. ODMG (Object Database Management Group):
     - Impedance mismatch is worse than the ozone hole!
  2. SQL:1999/2003:
     - Couldn’t care less – SQL rules!
     - We will discuss SQL only

Object Databases vs. Relational Databases

- **Relational**: set of relations; relation = set of tuples
- **Object**: set of classes; class = set of objects
- **Relational**: tuple components are primitive (int, string)
- **Object**: object components can be complex types (sets, tuples, other objects)
- **Unique features of object databases**:
  - Inheritance hierarchy
  - Object methods
    - Written in a standard language, e.g., C++ or Java
    - Similar to stored procedure, but stored procedures are not associated with particular relations
  - In some systems (ODMG), the host language and the data manipulation language are the same

Topics

- Motivation
  - Conceptual Object Data Model
  - Object Extensions in SQL
  - Objects in Oracle DBMS
  - Current Use of Objects in Databases
The Conceptual Object Data Model (CODM)

- Plays the same role as the relational data model
- Provides a common view of the different approaches (ODMG, SQL:1999/2003)

Object Id (Oid)

- Every object has a unique Id: different objects have different Ids
- Immutable: does not change as the object changes
- Different from primary key!
  - Like a key, identifies an object uniquely
  - But key values can change – oids cannot

Objects and Values

- An object is a pair: (oid, value)
- Example: A Joe Public’s object
  (#32, [ SSN: 111-22-3333,
    Name: “Joe Public”,
    PhoneN: {“516-123-4567”, “516-345-6789”},
    Child: {#445, #73} ]

Types of Values

- A value can be of one of the following types:
  - Primitive: an integer (eg, 7), a string (“John”), a float (eg, 23.45), a Boolean (eg, false)
  - Reference: An oid of an object, e.g., #445
  - Tuple: [A1: v1, …, An: vn]
    - A1, …, An – distinct attribute names
    - v1, …, vn – values
  - Set: {v1, …, vn}
    - v1, …, vn – values
- Complex value: reference, tuple, or set.
- Example: previous slide
Classes, Types & Extents

- **Class**: set of semantically similar objects (e.g., people, students, cars, motorcycles)
- A class has:
  - **Type**: describes common structure of all objects in the class (semantically similar objects are also structurally similar)
  - **Method signatures**: declarations of the operations that can be applied to all objects in the class.
  - **Extent**: the set of all objects in the class
- Classes are organized in a class hierarchy
  - The extent of a class contains the extent of any of its subclasses

Complex Types: Intuition

- Data (relational or object) must be properly structured
- **Complex data (objects) — complex types**
  - **Its type**: [SSN: String, Name: String, PhoneN: {String}, Child: {Person}]

Subtypes: Intuition

- A **subtype** has “more structure” than its supertype.
- **Example**: Student is a subtype of Person
  - **Person**: [SSN: String, Name: String, Address: [StNum: Integer, StName: String]]
  - **Student**: [SSN: String, Name: String, Address: [StNum: Integer, StName: String, Rm: Integer], Majors: {String}, Enrolled: {Course}]

Database Schema

- For each class includes:
  - **Type**
  - **Method signatures**: E.g., the following signature could be in class Course:
    - Boolean enroll(Student)
  - **The subclass relationship**
  - **The integrity constraints**
    - keys, foreign keys, etc.
Relation Schema (Ch 2_3.26)

- Relation name
- Attribute names & domains
- Integrity constraints like
  - The values of a particular attribute in all tuples are unique
  - The values of a particular attribute in all tuples are greater than 0
- Default values

Database Instance

- Set of extents for each class in the schema
- Each object in the extent of a class must have the type of that class
- Each object in the database must have unique oid
- The extents must satisfy the constraints of the database schema

Relational Database (Ch 2_3.27)

- Finite set of relations
- Each relation consists of a schema and an instance
- Database schema = set of relation schemas and constraints among relations (inter-relational constraints)
- Database instance = set of (corresponding) relation instances

Object-Relational Data Model

- A straightforward subset of CODM: only tuple types at the top level
- More precisely:
  - Set of classes, where each class has a tuple type (the types of the tuple component can be anything)
  - Each tuple is an object of the form (oid, tuple-value)
- Pure relational data model:
  - Each class (relation) has a tuple type, but
  - The types of tuple components must be primitive
  - Oids are not explicitly part of the model – tuples are pure values
Topics

• Motivation
• Conceptual Object Data Model
• Object Extensions in SQL
• Objects in Oracle DBMS
• Current Use of Objects in Databases

Objects in SQL:1999/2003

• Object-relational extension of SQL-92
• Includes the legacy relational model
• SQL:1999/2003 database = a finite set of relations
• relation = a set of tuples (extends legacy relations)
  OR
  a set of objects (completely new)
• object = (oid, tuple-value)
• tuple = tuple-value
• tuple-value = [Attr₁: v₁, ..., Attrₙ: vₙ]

SQL:1999 Tuple Values

• For attribute of tuples, in addition to the primitive types, constructed types are also supported
• Primitive types: integer, float, etc.
• Constructed types:
  – Row
  – Collection
  – User Defined Types or UDT
  – Reference Types or REFs

Row Types

• The same as the original (legacy) relational tuple type. However:
  – Row types can now be the types of the individual attributes in a tuple
  – Not in 1NF
  – In the legacy relational model, tuples could occur only as top-level types

CREATE TABLE PERSON (Name CHAR(20),
  Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)))


Row Types (Contd.)

- Use path expressions to refer to the components of row types:
  ```sql
  SELECT P.Name
  FROM    PERSON P
  WHERE  P.Address.ZIP = '11794'
  ```

- Update operations:
  ```sql
  INSERT INTO   PERSON(Name, Address)
  VALUES  ('John Doe', ROW(666, 'Hollow Rd.', '66666'))

  UPDATE  PERSON
  SET     Address.ZIP = '66666' 
  WHERE Address.ZIP = '55555'

  UPDATE  PERSON
  SET     Address = ROW(21, 'Main St', '12345') 
  WHERE Address = ROW(123, 'Maple Dr', '54321')  AND Name = 'J. Public'
  ```

User Defined Types (UDT)

- UDTs allow specification of complex objects/tuples, methods, and their implementation

- Like ROW types, UDTs can be types of individual attributes in tuples

- UDTs can be much more complex than ROW types (even disregarding the methods)
  - the components of UDTs do not need to be elementary types

A UDT Example

```sql
CREATE TYPE PersonType AS (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5))
);

CREATE TYPE StudentType UNDER PersonType AS {
    Id INTEGER,
    Status CHAR(2)
}

METHOD award_degree() RETURNS BOOLEAN;

CREATE METHOD award_degree() FOR StudentType
LANGUAGE C
EXTERNAL NAME 'file:/home/admin/award_degree';
```

Using UDTs in CREATE TABLE

- As an attribute type:
  ```sql
  CREATE TABLE TRANSCRIPT (Student StudentType, CrsCode CHAR(6), Semester CHAR(6), Grade CHAR(1))
  ```

- As a table type:
  ```sql
  CREATE TABLE STUDENT OF StudentType;
  ```
  Such a table is called typed table.
Objects

- Only typed tables contain objects (i.e., tuples with oids)
- Compare:
  
  ```
  CREATE TABLE STUDENT OF StudentType;
  and
  CREATE TABLE STUDENT1 ( 
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)),
    Id INTEGER,
    Status CHAR(2)
  )
  ```

- Both contain tuples of exactly the same structure
- Only the tuples in STUDENT—not STUDENT1—have oids
- Will see later how to reference objects, create them, etc.

Querying UDTs

- Nothing special—just use path expressions

```sql
SELECT T.Student.Name, T.Grade
FROM TRANSCRIPT T
WHERE T.Student.Address.Street = 'Main St.'
```

Note: T.Student has the type StudentType. The attribute Name is not declared explicitly in StudentType, but is inherited from PersonType.

Updating User-Defined Types

- Inserting a record into TRANSCRIPT:

```sql
INSERT INTO TRANSCRIPT(Student,Course,Semester,Grade)
VALUES (????, 'CS308', '2000', 'A')
```

The type of the Student attribute is StudentType. How does one insert a value of this type (in place of ????)?

Further complication: the UDT StudentType is encapsulated, i.e., it is accessible only through public methods, which we did not define

Do it through the observer and mutator methods provided by the DBMS automatically

Observer Methods

- For each attribute $A$ of type $T$ in a UDT, an SQL:1999 DBMS is supposed to supply an observer method, $A(\cdot) : T$, which returns the value of $A$ (the notation "\(\cdot\)" means that the method takes no arguments)
- Observer methods for StudentType:
  - Id (\cdot) \rightarrow INTEGER
  - Name (\cdot) \rightarrow CHAR(20)
  - Status (\cdot) \rightarrow CHAR(2)
  - Address (\cdot) \rightarrow ROW(INTEGER, CHAR(20), CHAR(5))
- For example, in

```sql
SELECT T.Student.Name, T.Grade
FROM TRANSCRIPT T
WHERE T.Student.Address.Street = 'Main St.'
```

Name and Address are observer methods, since T.Student is of type StudentType

Note: Grade is not an observer, because TRANSCRIPT is not part of a UDT, but this is a conceptual distinction—syntactically there is no difference
Mutator Methods

- An SQL DBMS is supposed to supply, for each attribute \( A \) of type \( T \) in a UDT \( U \), a mutator method \( A: T \rightarrow U \).

  For any object \( o \) of type \( U \), it takes a value \( t \) of type \( T \) and replaces the old value of \( o.A \) with \( t \); it returns the new value of the object. Thus, \( o.A(t) \) is an object of type \( U \).

- Mutators for \( \text{StudentType} \):
  - \( \text{Id} \): \( \text{INTEGER} \rightarrow \text{StudentType} \)
  - \( \text{Name} \): \( \text{CHAR(20)} \rightarrow \text{StudentType} \)
  - \( \text{Address} \): \( \text{ROW(INTEGER, CHAR(20), CHAR(5))} \rightarrow \text{StudentType} \)

Example: Inserting a UDT Value

```
INSERT INTO TRANSCRIPT(Student,Course,Semester,Grade)
VALUES (
  NEW StudentType()
  .Id(111111111)
  .Status('G5')
  .Name('Joe Public')
  .Address(ROW(123,'Main St.', '54321'))
  'CS532',
  'S2002',
  'A'
)
```

'CS532', 'S2002', 'A' are primitive values for the attributes Course, Semester, Grade

Create a blank StudentType object
Add a value for \( \text{Id} \)
Add a value for \( \text{Status} \)
Add a value for the Address attribute

Referencing Objects

- Consider again
  ```
  CREATE TABLE TRANSCRIPT (
    Student StudentType,
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade CHAR(1)
  )
  ```

- **Problem**: TRANSCRIPT records for the same student refer to distinct values of type \( \text{StudentType} \) (even though the contents of these values may be the same) – a maintenance/consistency problem

- **Solution**: use **self-referencing column** (next slide)
  - Bad design, which distinguishes objects from their references
  - Not truly object-oriented

Self-Referencing Column

- **Every typed table has a self-referencing column**
  - Normally invisible
  - Contains explicit object Id for each tuple in the table
  - Can be given an explicit name – the only way to enable referencing of objects

```
CREATE TABLE STUDENT2 OF \text{StudentType}
REF IS stud_oid;
```

Self-referencing columns can be used in queries just like regular columns
Their values cannot be changed, however
Reference Types and Self-Referencing Columns

• To reference objects, use self-referencing columns + reference types: REF(some-UDT)

CREATE TABLE TRANSCRIPT1 (  
  Student REF(StudentType) SCOPE STUDENT2,  
  CrsCode CHAR(6),  
  Semester CHAR(6),  
  Grade CHAR(1)  
)

Two issues:
• How does one query the attributes of a reference type
• How does one provide values for the attributes of type REF(…)
  – Remember: you can’t manufacture these values out of thin air – they are oids!

Querying Reference Types

• Recall: Student REF(StudentType) SCOPE STUDENT2 in TRANSCRIPT1.  
  How does one access, for example, student names?

  SQL:1999 has the same misfeature as C/C++ has (and which Java and OQL do not have):  
  it distinguishes between objects and references to objects.  
  To pass through a boundary of REF(…) use “Æ” instead of “.”

  SELECT T.StudentÆName, T.Grade  
  FROM TRANSCRIPT1 T  
  WHERE T.StudentÆAddress.Street = "Main St."

Inserting REF Values

• How does one give values to REF attributes, like Student in TRANSCRIPT1?
  • Use explicit self-referencing columns, like stud_oid in STUDENT2
  • Example: Creating a TRANSCRIPT1 record whose Student attribute has an object reference to an object in STUDENT2:

    INSERT INTO TRANSCRIPT1(Student,Course,Semester,Grade)  
    SELECT S.stud_oid, 'HIS666', 'F1462', 'D'  
    FROM STUDENT2 S  
    WHERE S.Id = '111111111'

Collection Data Types

• Support for arrays and multisets
  – Multiset (or bag): can have duplicates, unlike sets
  • Sets will look something like the following:

    CREATE TYPE StudentType UNDER PersonType AS (  
      Id INTEGER,  
      Status CHAR(2),  
      Enrolled REF(CourseType) MULTISET  
    )
Querying Collection Types

- For each student, list the Id, street, and the courses in which the student is enrolled:

  ```sql
  SELECT S.Id, S.Address, C.Name
  FROM STUDENT S, COURSE C
  WHERE C.CrsCode IN
    ( SELECT E.CrsCode
      FROM UNNEST(S.Enrolled) E
    )
  ```

  - Note: E is bound to tuples in a 1-column table of object references

Oracle 9i

- The following constructed types are supported:
  - Collection
  - User Defined Types or UDT
  - Reference Types or REFs
- The constructed type row is not supported

UDTs or Object Types

- An object type is a kind of datatype
  - Need to be defined
  - A value of an object type is an instance of that type
  - An object instance is also called an object
- Object types can have attributes and methods

```sql
CREATE TYPE person_type AS OBJECT (
  ssn NUMBER,
  name VARCHAR2(30),
  phone VARCHAR2(20)
) NOT FINAL;
```

Topics

- Motivation
- Conceptual Object Data Model
- Object Extensions in SQL
- Objects in Oracle DBMS
- Current Use of Objects in Databases
Relations With UDT Columns

• Can specify an object type as the datatype of a column in a relational table
  – Such objects that occupy table columns are called column objects
  -- An relational table with a column of object type

CREATE TABLE contacts (
  contact person_type,
  meeting DATE );

INSERT INTO contacts
VALUES(person_type(123, 'Joe', '504-222-3333'), sysdate);

Object Tables

• In an object table each row is an object
  -- An object table of person_type objects

CREATE TABLE persons OF person_type;

INSERT INTO persons
VALUES(person_type(123, 'Joe', '504-222-3333'));

INSERT INTO persons VALUES(199, 'John', '504-111-9222');

Object Tables (cont’d)

• Object table can be viewed as:
  – A single-column table in which each row is a person object
    • Allows performing object-oriented operations
  – A multi-column table
    • In which each attribute of the object type Person, occupies a column
    • Allows performing relational operations

Reference Types

• An Oracle built-in datatype
• A logical “pointer” to a row
  – Can use the dot notation to follow the pointers
  – Oracle does joins for you when needed
• Scoped REFs
  – Constrain the REF to a specified object table
    -- An relational table with reference to person objects

CREATE TABLE contacts2(
  contact REF person_type SCOPE IS persons,
  meeting DATE );
Collections

- Two collection datatypes:
  1. Varrays
     - Variable sized arrays
     - For ordered collections
  2. Nested tables:
     - Multisets
     - No order

Inheritance

- Subtypes can be created from types that are NOT FINAL
  - Structure and methods inherited by the subtype
  - Single inheritance only
- However, inheritance mechanism does not extend to subtables
  - i.e., class extent does not include subclass extent

```
CREATE TYPE student_type UNDER person_type(
  dept_id NUMBER,
  major VARCHAR2(30));
/
```

Topics

- Motivation
- Conceptual Object Data Model
- Object Extensions in SQL
- Objects in Oracle DBMS
  - Current Use of Objects in Databases

Pure Object vs. Object-Relational

- DBMS products are available which are built grounds-up around the object model. Lets call these OODBs
  - E.g., Objectivity/DB, ObjectStore
  - Very tight integration of DBMS with OO programming languages
- It is true (at least for now) that Object-relational have relegated OODBs to the sidelines
  - OODBs are niche products
Use of Object-Relational Features

- However, actual use of OR features, provided by major vendors, remains infrequent
- Reasons?
  - Lack of compatibility of features among vendors
  - SQL added extensive object support relatively recently
  - Conceptual design methodologies not as mature
    - When should I create object type vs. use pure relational with normalization?
    - Even with objects, myriad options in schema design. When to choose which design?
  - Performance tradeoffs not clear
- All of these problems could get mitigated with time