Object-Oriented Databases
and the ODMG Standard
Problems with Relational Databases

• Impedance mismatch between DML and application programming language:
  – different type systems
  – set processing vs. element-at-a-time processing
  – declarative vs. imperative programming paradigms.

• Normalization is a very bad idea:
  – an entity may be split over different tables
  – artificial keys are introduced when modeling nested sets

• Limited modeling power:
  – difficult to represent ordered sequences
  – difficult to represent hierarchical data (e.g., XML data)
  – little support for multimedia types (BLOBs)
  – no data abstraction (ADTs)

• Limited processing power:
  – cannot return a nested result
  – group-by cannot be perform over complex expressions/functions
OODBs

- *Object identity* identifies objects independent of value. Different from relational key.
- *Methods* model the behavior of an object.
- A *class* describes the state and the methods of every object that belongs to the class.
- Support for a complex type system, which includes atomic types (integer, real, string, etc), collection types (sets, bags, lists), records, etc.
  - types may be freely nested, which generalizes the nested relational model
  - classes define ADTs (new types and operations over these types)
- Support for *class inheritance*
The ODMG Standard

An OODB standard that supports:

• An Object Description Language (ODL) for schema specification (using classes).
• An Object Query Language (OQL) for querying data, which resembles SQL.
• Various host language bindings (C++, Java, Smalltalk, etc)

ODL supports various type constructors. A type constructor constructs complex types from simple ones. They can be freely composed:

• struct for building structures.
• various collection types, such as set, bag, and list.

    set< struct{ long x; list< struct{ string y; boolean z; list<string>; } >; } >;

The persistent root consists of the class extents (sets of class instances). In OQL you should refer to the extent, not to the class name. The class key uniquely identify objects within the class.
The University ODL Schema
module School {

struct Address { string city;
    short zipcode; };

class Person ( extent Persons key ssn )
{ attribute long ssn;
    attribute string name;
    attribute Address address;
};

...
typedef set<string> Degrees;

class Instructor extends Person ( extent Instructors )
{
    attribute long salary;
    attribute string rank;
    attribute Degrees degrees;
    relationship Department dept inverse Department::instructors;
    relationship set<Course> teaches inverse Course::taught_by;
    long works_for ();
    short courses ( in string dept_name );
};
class Department ( extent Departments keys dno, name )
{
    attribute long dno;
    attribute string name;
    attribute Instructor head;
    relationship set<Instructor> instructors inverse Instructor::dept;
    relationship set<Course> courses_offered inverse Course::offered_by;
    long avg_salary ();
};

class Course ( extent Courses keys code, name )
{
    attribute string code;
    attribute string name;
    relationship Department offered_by inverse Department::courses_offered;
    relationship Instructor taught_by inverse Instructor::teaches;
    relationship set<Course> is_prerequisite_for inverse Course::has_prerequisites;
    relationship set<Course> has_prerequisites inverse Course::is_prerequisite_for;
};
## OQL

### Path expressions:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.A</td>
<td>If x is an object or a struct and A is an attribute or a relationship name</td>
</tr>
<tr>
<td>x.m(e1,…,en)</td>
<td>If x is an object and m is a method</td>
</tr>
</tbody>
</table>

For example, if e: Instructor, d: Department, c: Course, then
- e.dept.name
- d.head.name
- d.instructors
- c.taught_by.dept.name

Illegal path expression:
- d.instructors.name (since d.instructors is a set<Instructor>)
struct ( a: 1, b: "a" )
struct ( x: struct( w: 1.7, a: “aa”), y: 23 )
Department( name: “cse”, dno: 1289 )
Instructor( name: “Jones”, ssn: 2345, salary: 8000, dept: d, degrees: set(“MS”, “PhD”) )
Constant Sets, Bags, and Lists

set(1,2,1) (set of long)
bag(“aaa”, “b”, “aaa”, “b”) (set of string)
set(set(1,2), set(2,1)) (set of set of long)
element(set(3)) → 3
list(1,2)[1] → 2
list(list(1,2), list(3,4))[1][0] → 3
last(list(1,2,3)) → 3
first(list(1,2,3)) → 1
max(set(1,2,3)) → 3
Select-from-where

**Remember:** 1) entry points must be extent names 2) values in the FROM clause must be expressions that evaluate to collections.
**Rule:** if a value is a collection, you use it in a FROM clause, otherwise, you use it in a path expression.

```sql
select ssn, salary
from Instructors
where name = "Smith"

select struct (issn: ssn, isalary: salary )
from Instructors
where ssn>10 and ssn<12

select distinct e.dept.name
from Instructors e
where e.dept.head.name = "Smith"
```
Dependent Joins

```sql
select c.code
from e in Instructors,
     c in e.teaches
where e.name = "Smith"

select c.name
from c in Courses,
     d in c.is_prerequisite_for
where d.name = "Advanced Databases"

select ename: e.name, dname: e.dept.name
from e in Instructors,
     s in e.dept.instructors,
     c in e.teaches
where s.name = "Smith" and c.name = "DB II"
```
Types are Very Important

select ename: e.name, dname: e.dept.name
from e in Instructors, (e is of type Instructor)
    s in e.dept.instructors, (s is of type Instructor)
    c in e.teaches (c is of type Course)
where s.name = “Smith” and c.name = “DB II”

So s.name, c.name, and e.name are of type string, e.dept is of type Department, and e.dept.name is of type string.

The result type is:

    bag(struct(ename:string,dname:string))
Each relationship gives you two ways of accessing data:

<table>
<thead>
<tr>
<th>Left Code</th>
<th>Right Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>select c.code from e in Instructors, c in e.teaches where e.name = “Smith”</td>
<td>select c.name from c in Courses, d in c.is_prerequisite_for where d.name = “Advanced Databases”</td>
</tr>
<tr>
<td>select c.code from c in Courses where c.taught_by.name = “Smith”</td>
<td>select c.name from d in Courses, c in d.has_prerequisites where d.name = “Advanced Databases”</td>
</tr>
</tbody>
</table>
Other OQL Constructs

The order-by clause constructs an ordered list and select-distinct constructs a set. Exists and for-all are used for existential and universal quantification:

select e
from e in Instructors
where e.ssn > 10 and e.ssn < 15
order by e.name

(select e from e in Instructors order by e.name)[10]
for all x in Instructors: x.salary >= 10000
exists x in Instructors: x.salary >= 50000

exists e in Instructors:
    for all c in e.teaches:
        e.ssn > 10 and c.name = “DB II”

element(select * from Instructors where ssn=60).teaches
Nested Queries

```sql
select e.ssn, e.salary
from e in (select n from n in Instructors
    where n.name = "Smith")

select e.ssn
from e in Instructors
where exists c in e.teaches: c.offered_by.name = e.dept.name
    and e.name = "Smith"

flatten(select teaches from Instructors where name = "Smith")

sum( select sum(select e.salary from e in d.instructors)
    from d in Departments )
```
More Nested Queries

```sql
select count(c.has_prerequisites)
from c in Courses
where c.name = "DB II"

select x: e.name, y: (select c.name from c in e.teaches)
from e in Instructors
where e.ssn = 50

select x: e.name, y: (select x: c.name, y: count(c.has_prerequisites)
    from c in e.teaches)
from e in Instructors
where e.ssn = 50
```
Group-By

select distinct rank: er, avg_salary: avg(e.salary)
from e in Instructors
group by er: e.rank

select distinct dno: dn, instr_num: count(e)
from e in Instructors
group by dn: e.dept.dno
having dn > 6

select x, y, c: count(c)
from e in Instructors, c in e.teaches
group by x: e.ssn, y: c.name
having x > 60 and y > “DB”
More Group-By

select distinct x, y, c: count(e)
from e in Instructors
group by x: count(e.teaches),
          y: (exists c in e.teaches: c.name="DB II")
having x>0

Type of output: set(struct(x:long,y:boolean,c:long))

select distinct dno, low, high, c: count(d)
from d in Departments
group by dno: d.dno,
       low: count(d.courses_offered) <10,
       high: count(d.courses_offered) >=10
The \textbf{partition} Variable

More complex group-bys require the use of the \textbf{partition} variable:

\begin{verbatim}
select degrees, prereqs, count(partition),
    max(select p.e.salary from p in partition)
from e in Instructors,
    c in e.teaches
group by degrees: count(e.degrees),
    prereqs: count(c.has_prerequisites)
\end{verbatim}

Partition is a set(struct(e: Instructor,c: Course)) and contains the (e,c) pairs for a single group.