Object Databases

CP465 Databases II

Dr. Ilias S. Kotsireas

ikotsire@wlu.ca

Wilfrid Laurier University

1. Object Identity
2. Object Structure
3. Type Constructors
4. Encapsulation of:
   (a) Operations
   (b) Methods
   (c) Persistence

Object Identity

An ODMS provides a **unique identity** to each independent object stored in the database.

This feature is implemented via a system-generated **object identifier (OID)**.

Values of OIDs are not visible to the external user.

OIDs are used internally by the system to identify each object uniquely and to create/manage references between objects.

OIDs are **immutable** (don't change over time)

OIDs should be **used only once** (even if the object is removed, its OID is not used)

Therefore OIDs should not depend on:
(a) object attribute values
(b) physical address of the object in storage

Long integers are commonly used as OIDs.
A hash table is used to map the OID values to the physical address.

Object Structure

Most ODMS allow for the representation of **objects** and **values**.

Objects have OIDs. Values do not have OIDs.

A value is typically stored within an object and cannot be referenced from other objects.

Some systems allow for **structured values**.

In Object Databases the state (current value) of a object can be constructed from other objects/values using certain **type constructors**.

Formal way to represent such objects: triplet

(i, c, v) = (OID, type constructor, object state/value)

v is interpreted based on the constructor c.

Type constructors in the Object Model:
atom, **tuple**, **set**, **list**, **bag**, **array**.

The atom constructor is used to represent all basic atomic values: integers, reals, strings, booleans.

- c = atom, v = atomic value
- c = set, v = $\{i_1, \ldots, i_n\}$ set of OIDs of objects of the same type
- c = tuple, v = $< a_1 : i_1, \ldots, a_n : i_n >$ instance variable (attribute name) : OID
- c = list, v = $[i_1, \ldots, i_n]$ ordered list of OIDs of objects of the same type
- c = array, v = 1-dimensional array of OIDs, (has a max. # of elements)

Difference between sets and bags:
duplication is allowed (multi-sets)

Arbitrary nesting of type constructors is allowed

The state of an object that is not of type atom, refers to other objects by their OIDs.

The only case where an actual value appears is in the state of an object of type atom.

Example of a complex object

$o_1 = (i_1, atom, "London")$
$o_2 = (i_2, atom, "Waterloo")$
$o_3 = (i_3, atom, "Hamilton")$
$o_4 = (i_4, atom, 5)$
$o_5 = (i_5, atom, "Research")$
$o_6 = (i_6, atom, "November 20, 2000")$
$o_7 = (i_7, set, \{i_1, i_2, i_3\})$
$o_8 = (i_8, tuple, < dname : i_5, dnum : i_4, manager : i_9,$
$locations : i_7, employees : i_{10}, projects : i_{11} >)$
$o_9 = (i_9, tuple, < manager : i_{12}, mgrStartDate : i_6 >)$
$o_{10} = (i_{10}, set, \{i_{12}, i_{13}, i_{14}\})$
$o_{11} = (i_{11}, set, \{i_{15}, i_{16}, i_{17}\})$
$o_{12} = (i_{12}, tuple, < fname : i_{18}, minit : i_{19}, lname : i_{20},$
$SIN : i_{21}, \ldots, salary : i_{26}, dpt : i_8 >)$

- $o_1 - o_6$ represent atomic values
- o_7 set-valued object, represents the set of locations for dpt. 5
- o_8 tuple-valued object, represents dpt. 5 (some attributes have atomic values and some not)
- o_{12} represents an employee of dpt. 5 Research

An object can be represented as a graph structure that can be constructed by applying the type constructors recursively:

1. create a node for each object o_i

2. label each node with the OID and type constr. for object o_i

3. create a node for each basic atomic value (and label it with this value)

4. if o_i has an atomic value, draw a directed arc from the node representing o_i to the node representing this value

5. if o_i has a constructed value, draw a directed arc from the node representing o_i to a node represented the constructed value

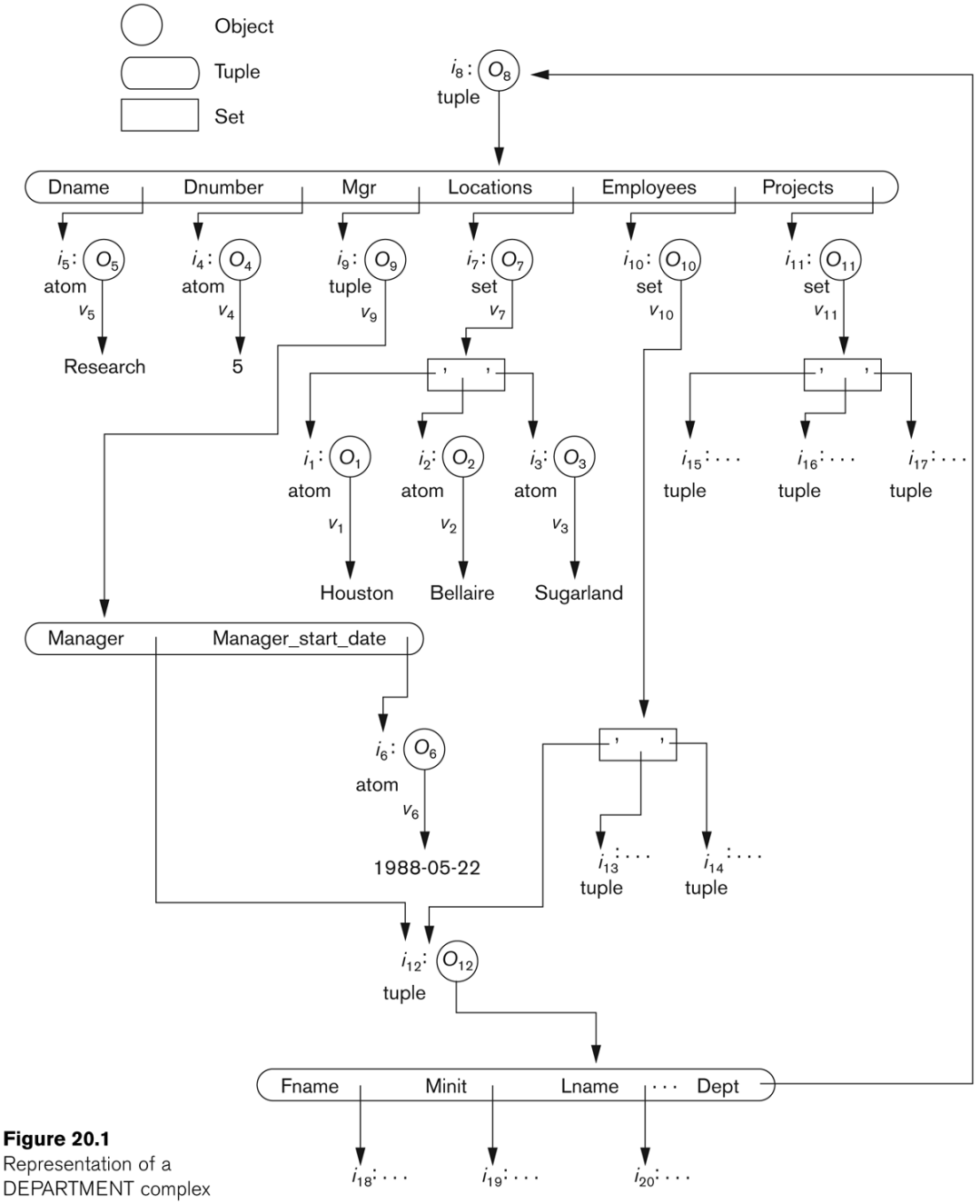

Figure 20.1
Representation of a DEPARTMENT complex object as a graph.

Object Equality

(1) two objects are said to have **identical states** (deep equality), if the graphs representing their states are identical, including same OIDs at every level.

(2) two objects are said to have **equal states** (shallow equality), if the graphs representing their states have identical structure, some internal nodes can have objects with different OIDs.

Example: Identical vs Equal Objects

$o_1 = (i_1, tuple, < a_1 : i_4, a_2 : i_6 >)$
$o_2 = (i_2, tuple, < a_1 : i_5, a_2 : i_6 >)$
$o_3 = (i_3, tuple, < a_1 : i_4, a_2 : i_6 >)$
$o_4 = (i_4, atom, 10)$
$o_5 = (i_5, atom, 10)$
$o_6 = (i_6, atom, 20)$

The objects o_1 and o_2 have equal states,
but not identical states.

The objects o_1 and o_3 have identical states.

Type Constructors

ODL is used to define the object types for a particular database.

The type constructors are used to define the data structures for an object database schema.
(definitions of operations/methods go into the schema as well)

Attributes that refer to other objects are **references** to other objects and serve to represent **relationships** among the object types.

A binary relationship can be represented in one direction, or it can have an **inverse reference**.

Example

The attribute Employees of Department has as its value a set of Employee OIDs.

The attribute Department of Employee is the inverse reference.

ODMG allows inverses to be explicitly declared, to ensure consistency.

```
define type EMPLOYEE
    tuple ( Fname:          string;
            Minit:          char;
            Lname:          string;
            Ssn:            string;
            Birth_date:     DATE;
            Address:        string;
            Sex:            char;
            Salary:         float;
            Supervisor:     EMPLOYEE;
            Dept:           DEPARTMENT;

define type DATE
    tuple ( Year:           integer;
            Month:          integer;
            Day:            integer; );

define type DEPARTMENT
    tuple ( Dname:          string;
            Dnumber:        integer;
            Mgr:            tuple ( Manager:    EMPLOYEE;
                                    Start_date: DATE;     );
            Locations:      set(string);
            Employees:      set(EMPLOYEE);
            Projects        set(PROJECT);    );
```

Figure 20.2
Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

Encapsulation of Operations, Methods, Persistence

Main Idea define the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type.

The internal structure of the object is hidden.

The object is accessible only through a number of predefined operations:
creation, destruction, update object state, retrieve parts of the object state, perform a calculation.

The **implementation** of an operation can be specified in a programming language.

External users of the object can only see the **interface** of the object type, which defines the name & arguments of each operation.

The implementation is hidden from the external users.

signature the interface part of each operation

method the operation implementation

A method is invoked by sending a **message** to the object, that will execute the corresp. method.

Part of the execution may involve sending another message to another object.

Problem The requirement that all objects are completely encapsulated is too stringent, for database applications.

Solution Divide the structure of the object into **visible** and **hidden** attributes (instance variables).

The visible attributes can be accessed via OQL. The hidden attributes can be accessed by predefined operations only.

class object type definition + definitions of operations for that type

Typical operations: **object constructor**, **object destructor**, **object modifier operations**, **retrieval**

dot notation
apply an operation to an object, refer to an attrib.

```
define class EMPLOYEE
   type tuple (   Fname:        string;
                  Minit:        char;
                  Lname:        string;
                  Ssn:          string;
                  Birth_date:   DATE;
                  Address:      string;
                  Sex:          char;
                  Salary:       float;
                  Supervisor:   EMPLOYEE;
                  Dept:         DEPARTMENT;   );
   operations     age:          integer;
                  create_emp:   EMPLOYEE;
                  destroy_emp:  boolean;
end EMPLOYEE;

define class DEPARTMENT
   type tuple (   Dname:        string;
                  Dnumber:      integer;
                  Mgr:          tuple (   Manager:      EMPLOYEE;
                                          Start_date:   DATE;     );
                  Locations:    set(string);
                  Employees:    set(EMPLOYEE);
                  Projects      set(PROJECT);    );
   operations     no_of_emps:   integer;
                  create_dept:  DEPARTMENT;
                  destroy_dept: boolean;
                  assign_emp(e: EMPLOYEE): boolean;
                  (* adds an employee to the department *)
                  remove_emp(e: EMPLOYEE): boolean;
                  (* removes an employee from the department *)
end DEPARTMENT;
```

Figure 20.3
Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

Object Persistence via Naming and Reachability

An ODMS is used in conjunction with an OO progr. lang.

An object is typically created by an application program, by invoking the object constructor operation.

Not all objects are meant to be stored permanently in the database.

Transient objects exist in the executing program and are lost when the program execution terminates.

Persistent objects are stored in the database and stay there after program execution terminates.

Typical mechanisms for achieving object persistence: **Naming** and **Reachability**.

Naming

giving an object a unique persistent name through which it can be retrieved by a program.

the named persistent objects are used as **entry points** to the database. users and applications can start their database access.

for large databases with thousands of objects, it's impractical to give names to all objects.

Reachability

most objects are made persistent by making them reachable from some persistent object.

object B is reachable from object A, if a sequence of references in the object graph lead from A to B.

if we create a named persistent object N, whose state is a set/list of objects of some type T, then we can make objects of type T persistent by simply adding them to the set/list.
(making them reachable from N)

```
define class DEPARTMENT_SET:
    type set (DEPARTMENT);
    operations add_dept(d: DEPARTMENT): boolean;
            (* adds a department to the DEPARTMENT_SET object *)
                remove_dept(d: DEPARTMENT): boolean;
            (* removes a department from the DEPARTMENT_SET object *)
                create_dept_set:    DEPARTMENT_SET;
                destroy_dept_set:   boolean;
end DepartmentSet;
. . .

persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
(* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
. . .

d:= create_dept;
(* create a new DEPARTMENT object in the variable d *)
. . .

b:= ALL_DEPARTMENTS.add_dept(d);
(* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

Figure 20.4
Creating persistent objects by naming and reachability.