Object Databases – Concepts

Introduction

- Traditional Data Models:
  - Hierarchical
  - Network (since mid-60's)
  - Relational (since 1970 and commercially since 1982)
- Object Oriented (OO) Data Models since mid-90's
- Reasons for creation of Object Oriented Databases
  - Need for more complex applications.
  - Need for additional data modeling features.
  - Increased use of object-oriented programming languages.
- Commercial OO Database products – ObjectStore, O2

OO Databases – main claim

- OO databases try to maintain a direct correspondence between real-world and database objects.
- Hence objects do not lose their integrity and identity and can easily be identified and operated upon.
- Objects may have an object structure of arbitrary complexity in order to contain all of the necessary information that describes the object.

OO Concepts

- Object Identity
- Object Structure
- Type Constructors
- Object Persistence
- Type and Class Hierarchy
- Complex Objects
- Polymorphism
- Multiple Inheritance

Object Identity

- An OO database system provides a unique identity to each independent object stored in the database.
- This unique identity is typically implemented via a unique, system-generated **object identifier**, or **OID.**
- OID – not visible to user, but used internally by system to identify each object.
- Two important properties:
  - OID is **immutable –** value for an object should not change.
  - Each OID be used only once – even if an object is removed, its OID should not be assigned to another object.

Object Identity

- OID should not depend on any attribute values of the object.
- OO databases allow for the representation of both objects and values.
- A value is typically stored within an object and cannot be referenced from other objects.

Object Structure

- In OO databases, the state (current value) of a complex object may be constructed from other objects (or other values) by using certain type constructors.
- An object is defined by a triple (OID, type constructor, state).
- The three most basic constructors are **atom**, **tuple**, and **set**.
- Other commonly used constructors include **list**, **bag**, and **array**.
- The atom constructor is used to represent all basic atomic values.
  - such as integers, real numbers, character strings, Booleans, and any other basic data types.

Object Structure

- The object state is interpreted based on the constructor c.
- If c= atom, the state is an atomic value from domain of basic values.
- If c = set, the state is a set of object identifiers – OIDs $\{i_1,i_2,....i_n\}$ that are of same type.
- If c = tuple, then the state is tuple of form $<a_1:i_1, a_2:i_2,......, a_n:i_n>$ where each a_i is an attribute name and each i_j is an OID.
- A list is similar to a set except that the OIDs in a list are ordered.
- If c=array, the state of the object is a single-dimensional array of OID.
- The difference between set and bag is all elements in a set must be distinct whereas a bag can have duplicate elements.

Example 1: Complex Object

- We use i_1, i_2, i_3, . . . to stand for unique system-generated object identifiers. Consider the following objects:

o_1 = (i_1, atom, 'Houston')

o_2 = (i_2, atom, 'Bellaire')

o_3 = (i_3, atom, 'Sugarland')

o_4 = (i_4, atom, 5)

o_5 = (i_5, atom, 'Research')

o_6 = (i_6, atom, '1988-05-22')

o_7 = (i_7, set, {i1, i2, i3})

Example 1: Complex Object

- Consider the following objects: (cont...)

o_8 = (i_8, tuple, <dname:i_5, dnumber:i_4, mgr:i_9, locations:i_7, employees:i_{10}, projects:i_{11}>)

o_9 = (i_9, tuple, <manager:i_{12}, manager_start_date: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}, ssn:i_{21}, . . ., salary:i_{26}, supervisor:i_{27}, dept:i_8>)

…..

…..

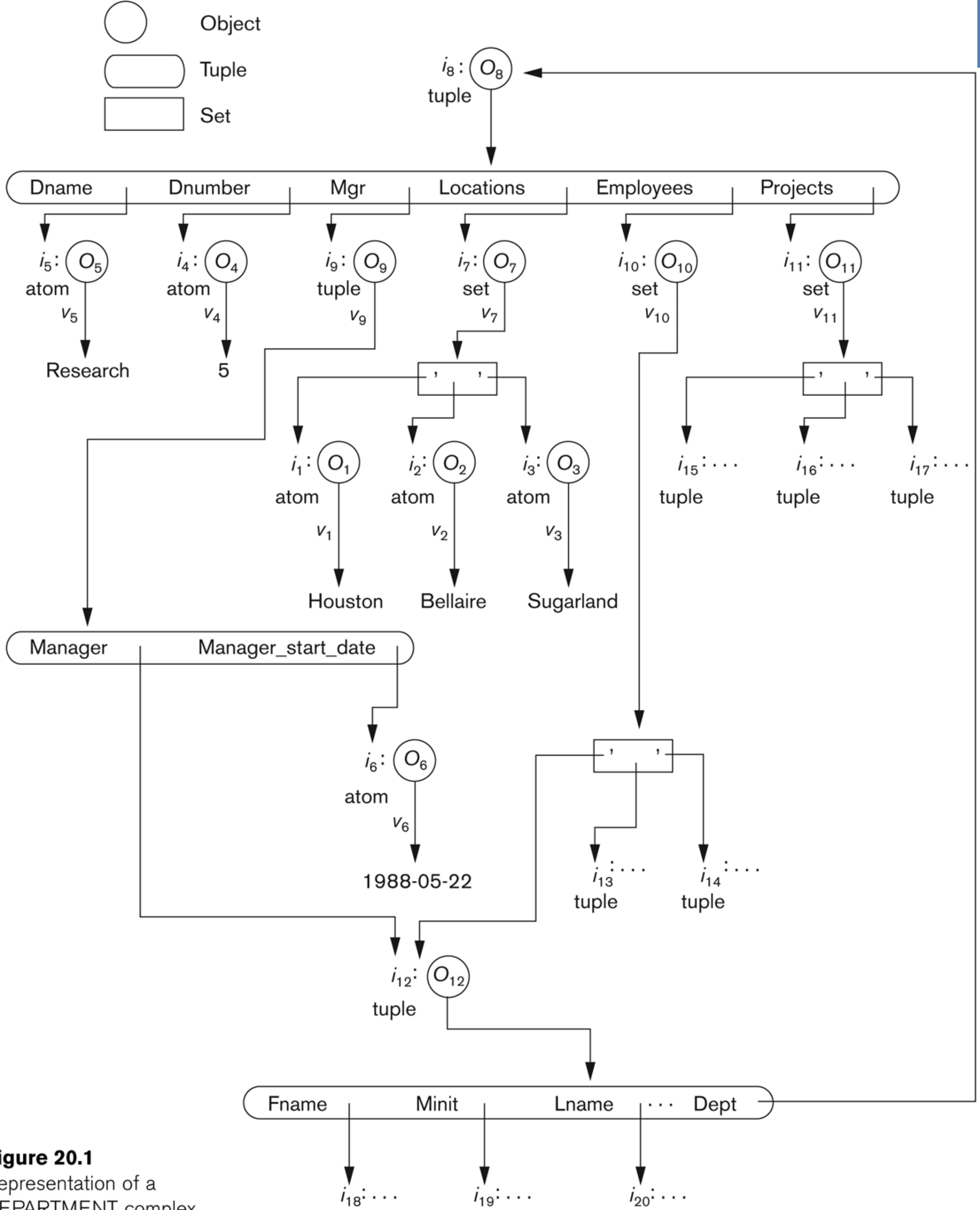

Figure 20.1
Representation of a DEPARTMENT complex object as a graph.

Example 2: Identical vs Equal Objects

- Two objects are said to have **identical** state, if its states are identical, including the OIDs.
- Two objects are said to have **equal** state, if its states are identical, but values are reached through objects with different OIDs.

Example 2: Identical vs Equal Objects

- Consider the following objects:

  $o_1 = (i_1, \text{tuple}, \langle a_1{:}i_4, a_2{:}i_6\rangle)$

  $o_2 = (i_2, \text{tuple}, \langle a_1{:}i_5, a_2{:}i_6\rangle)$

  $o_3 = (i_3, \text{tuple}, \langle a_1{:}i_4, a_2{:}i_6\rangle)$

  $o_4 = (i_4, \text{atom}, 10)$

  $o_5 = (i_5, \text{atom}, 10)$

  $o_6 = (i_6, \text{atom}, 20)$

- The states of objects o_1 and o_2 are *equal*.
- The states of objects o_1 and o_3 are *identical*.

Type Constructors

```
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.

Object Behavior via Class Operations

- The **behavior** of a type of object can be defined based on the **operations** that can be externally applied to objects of that type.
- The internal structure of object is hidden, and is accessible only through a predefined operations.
- The **implementation** of an operation can be specified in a *general-purpose programming language*.
- The term class refers to an object type definition, along with the operation definitions.

Object Behavior via Class Operations

```
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 Behavior via Class Operations

- A number of operations are declared for each class.
- Typical operation includes:
  - Object constructor – to create a new object.
  - Object destructor – to destroy an object.
  - Object modifier – to modify the states of various attributes of an object.

Object Persistence

- Transient objects – exists in executing program and disappears once the program terminates.
- Persistent objects – stored in database and persist after program termination.
- Mechanism for making an object persistent are:
  - Naming
  - Reachability
- Naming mechanism – Assign an object a unique persistent name through which it can be retrieved by this and other programs.
- Named persistent objects are used as entry points to the database.

Object Persistence

- Not practical to give names to all objects in a large databases.
- Solution – reachability mechanism.
- Make the object reachable from some persistent object.
- An object B is said to be **reachable** from an object A if a sequence of references in the object graph lead from object A to object B.
- In traditional database models such as relational model or EER model, all objects are assumed to be persistent.
- In OO approach, a class declaration specifies only the *type* and *operations* for a class of objects.

Object Persistence

- The user must separately define a **persistent object** of **type set** (DepartmentSet) or **list** (DepartmentList) whose value is the collection of references to all persistent DEPARTMENT objects

Object Persistence

```
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.

Type and Class Hierarchy

- A type can be defined by giving it a type name and then listing the names of its visible (**public**) functions.
- When specifying a type, use the following format:
  - TYPE_NAME: function, function, . . . , function
- Example:
  - PERSON: Name, Address, Birthdate, Age, Ssn.
- To create a new type that is similar but not identical to an already defined type – concept is subtype.
- Then the subtype inherits all the functions of the predefined type.

Type and Class Hierarchy

- Example:

  EMPLOYEE subtype-of PERSON: Salary, Hire_date, Seniority

  STUDENT subtype-of PERSON: Major, GPA

- Consider a type that describes objects in plane geometry:

  GEOMETRY_OBJECT: Shape, Area, ReferencePoint

- Define a number of subtypes for the GEOMETRY_OBJECT type:

  RECTANGLE **subtype-of** GEOMETRY_OBJECT: Width, Height

  TRIANGLE **subtype-of** GEOMETRY_OBJECT: Side1, Side2, Angle

  CIRCLE **subtype-of** GEOMETRY_OBJECT: Radius

Extents

- Extents are collections of objects of the same type.
- **Persistent Collection**:
- This holds a collection of objects that is stored permanently in the database and can be accessed and shared by multiple programs.
- **Transient Collection**:
- This exists temporarily during the execution of a program but is not kept when the program terminates.

Complex Objects

- The principal motivation for development of OO systems was to represent complex objects.
- Two main types of complex objects – structured and unstructured.
- Unstructured complex object:
  - Facility provided by DBMS to store and retrieve large objects.
  - Not part of standard data types in traditional db.
  - The db does not have the capability to query and other operations on values of these objects.
  - Example: BLOBs, CLOBs

Complex Objects

- Unstructured complex object:
  - In OODBMS, new abstract data type can be defined and methods for selecting, comparing, and displaying such object can be provided for uninterpreted object.
  - OODBMS allows users to create new types that includes both structure and operations.
  - Libraries of new types can be defined that can be later used or modified by creating subtypes of the types provided in library.

Complex Objects

- Object's structure is defined by repeated application of the type constructors.
- Two types of reference semantics:
- Ownership semantics – Dname, Dnumber, Mgr, Locations
  - Called is-part-of relationship.
  - Component objects are encapsulated within complex object.
  - No need to have OID, and can be accessed only by methods.
  - Deleted if the object itself is deleted.
- Reference semantics – Employee, Projects
  - Called is-associated-with relationship.

Complex Objects

- Reference semantics – Employee, Projects
  - Referenced components are independent objects that have their own identity and methods.
  - The complex object must invoke appropriate method to access the referenced component.
  - The referenced component may be referenced by more than one complex object.
  - Hence not deleted when complex object is deleted.

Complex Obje

- DEPARTMENT object:

  1 level: two basic values, four complex structure.

  2 level: one tuple structure, three set structure.

  ….

  ….

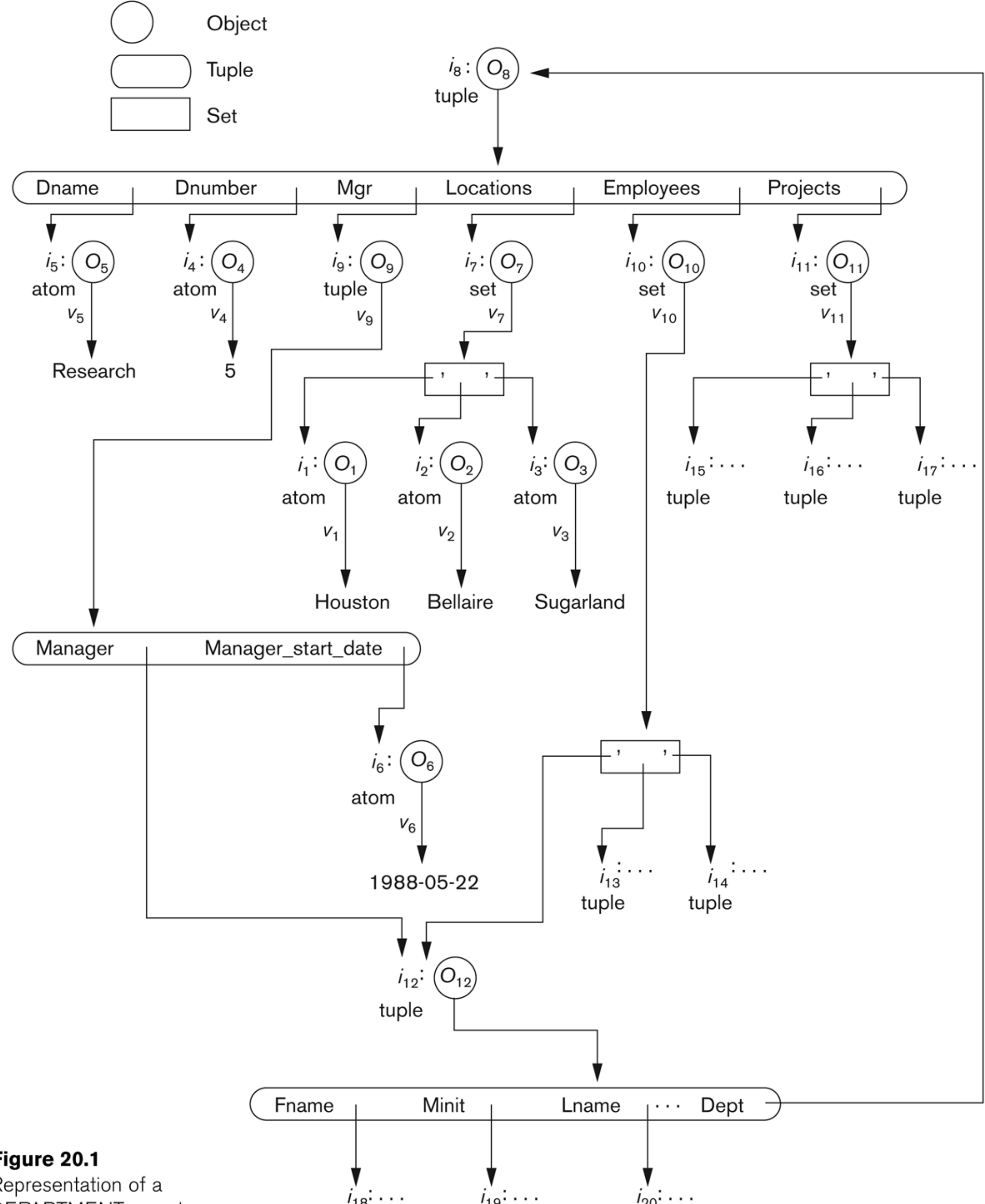

Figure 20.1
Representation of a DEPARTMENT complex object as a graph.

Polymorphism

- This concept allows the same **operator name or symbol** to be bound to two or more **different implementations** of the operator, depending on the type of objects to which the operator is applied
- Ex: GEOMETRY_OBJECT: Shape, Area, ReferencePoint

  RECTANGLE **subtype-of**

  GEOMETRY_OBJECT (Shape='rectangle'): Width, Height

  TRIANGLE **subtype-of**

  GEOMETRY_OBJECT (Shape='triangle'): Side1, Side2, Angle

  CIRCLE **subtype-of** GEOMETRY_OBJECT (Shape='circle'): Radius
- The database must select appropriate method for *Area* function based on type of geometric object applied.

Multiple Inheritance

- Multiple inheritance occurs when a certain subtype T is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes.
- For example, we may create a subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER – forms lattice.
- Issue: if both MANAGER and ENGINEER have function Salary and is implemented by different methods, then ambiguity is to which of two is inherited by subtype ENGINEERING_MANAGER.
- Some OO systems do not permit multiple inheritance at all.

References

- Fundamentals of Database Systems, Ramez Elamsri, Navathe.