Object Databases

Chapter 14

1

What's in This Module?

- Motivation
- Conceptual model
- SQL:1999/2003 object extensions
- ODMG
  - ODL – data definition language
  - OQL – query language
- CORBA

2

Problems with Flat Relations

Consider a relation

Person(*SSN*, *Name*, *PhoneN*, *Child*)

with:

- FD: *SSN* → *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

3

An Instance of Person

SSN	*Name*	*PhoneN*	*Child*
111-22-3333	Joe Public	516-123-4567	222-33-4444
111-22-3333	Joe Public	516-345-6789	222-33-4444
111-22-3333	Joe Public	516-123-4567	333-44-5555
111-22-3333	Joe Public	516-345-6789	333-44-5555
222-33-4444	Bob Public	212-987-6543	444-55-6666
222-33-4444	Bob Public	212-987-1111	555-66-7777
222-33-4444	Bob Public	212-987-6543	555-66-7777
222-33-4444	Bob Public	212-987-1111	444-55-6666

4

Dependencies in Person

Join dependency (JD):

Person = $(SSN, Name, PhoneN) \bowtie (SSN, Name, Child)$

Functional dependency (**FD**):

$SSN \rightarrow Name$

5

Redundancies in **Person**

- Due to the JD:

  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

- Due to the FD:

  Joe Public is associated with the SSN 111-22-3333 four times (for each of Joe's child and phone)!

  Similarly for Bob Public

6

Dealing with Redundancies

- What to do? *Normalize*!
  - Split Person according to the JD
  - Then each resulting relation using the FD
  - Obtain four relations (two are identical)

7

Normalization removes redundancy:

Person1

SSN	*Name*
111-22-3333	Joe Public
222-33-4444	Bob Public

Phone

SSN	*PhoneN*
111-22-3333	516-345-6789
111-22-3333	516-123-4567
222-33-4444	212-987-6543
222-33-4444	212-135-7924

ChildOf

SSN	*Child*
111-22-3333	222-33-4444
111-22-3333	333-44-5555
222-33-4444	444-55-6666
222-33-4444	555-66-7777

8

But querying is still 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.*PhoneN*

FROM Person1 P, ChildOf C, ChildOf G, Phone N

WHERE P.*Name* = 'Joe Public' AND P.*SSN* = C.*SSN* AND

C.*Child* = G.*SSN* AND G.*Child* = N.*SSN*

9

Objects Allow Simpler Design

Schema:

Person(*SSN*: String,
Name: String,
PhoneN: **{**String**}**,
Child: **{**SSN**}**)

Set data types

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}
)

10

Objects Allow Simpler Queries

Schema (slightly changed):
Person(*SSN*: String,
Name: String,
PhoneN: {String},
Child: **{**Person**}**)

Set of persons

- 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'

Path expression

- Much more natural!

11

ISA (or Class) 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

12

Object Methods in Queries

- Objects can have associated operations (methods), which 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:

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

13

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**: Set-oriented, works with relations, uses high-level operations over them.
- **Host language**: Record-oriented, does not understand relations and high-level operations on them.
- **SQL**: Declarative.
- **Host language**: Procedural.
- Embedding SQL in a host language involves ugly adaptors (cursors/iterators) – a direct consequence of the above mismatch of properties between SQL and the host languages. It was dubbed *"impedance" mismatch*.

14

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 s 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.)

15

Is Impedance Mismatch Really a Problem?

- The jury is out
- Two main approaches/standards:
  - ODMG (Object Database Management Group):

    Impedance mismatch is worse that the ozone hole!
  - SQL:1999/2003:

    Couldn't care less – SQL rules!
- We will discuss both approaches.

16

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
  – In some systems (ODMG), the host language and the data manipulation language are the same

17

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)
- Close to the ODMG model, but is not burdened with confusing low-level details

18

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

19

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}])

20

Complex Values

- A *value* can be of one of the following forms:
  - *Primitive* value: an integer (eg, 7), a string (“John”), a float (eg, 23.45), a Boolean (eg, *false*)
  - *Reference* value: An oid of an object, e.g., #445
  - *Tuple* value: $[A_1: v_1, \ldots, A_n: v_n]$
    - $A_1, \ldots, A_n$ – distinct attribute names
    - $v_1, \ldots, v_n$ – values
  - *Set* value: $\{v_1, \ldots, v_n\}$
    - $v_1, \ldots, v_n$ – values
- *Complex value*: reference, tuple, or set.
- Example: previous slide

21

Classes

- *Class*: set of semantically similar objects (eg, 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*

22

Complex Types: Intuition

- Data (relational or object) must be properly structured
- Complex data (objects) – complex types

Object: (#32, [*SSN*: 111-22-3333,
Name: "Joe Public",
PhoneN: {"516-123-4567", "516-345-6789"},
Child: {#445, #73}])

Its type: [*SSN*: String,
Name: String,
PhoneN: {String},
Child: {Person}]

23

Complex Types: Definition

- A *type* is one of the following*:*
  - *Basic* types: String, Float, Integer, etc.
  - *Reference* types: user defined class names, *eg*, **Person**, **Automobile**
  - *Tuple* types: [A_1: T_1, .., A_n: T_n]
    - A_1, .., A_n – distinct attribute names
    - T_1, .., T_n – types
    - *Eg*, [*SSN*: String, *Child*: {Person}]
  - *Set* types: {*T*}, where *T* is a type
    - *Eg*, {String}, {Person}
- *Complex* type: reference, tuple, set

24

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}]

25

Subtypes: Definition

- T is a *subtype* of T' iff $T \neq T'$ and
  - *Reference types:*
    T, T' are *reference* types and T is a subclass T'
  - *Tuple types:*
    $T = [A_1: T_1, .., A_n: T_n, A_{n+1}: T_{n+1}, .., A_m: T_m]$,
    $T' = [A_1: T_1', .., A_n: T_n']$
    are *tuple types* and for each $i=1,\ldots,n$, either $T_i = T_i'$ or T_i is a subtype of T_i'
  - *Set types:*
    $T = \{T_0\}$ and $T' = \{T_0'\}$ are *set types* and T_0 is a subtype of T_0'

26

Domain of a Type

- *domain*(T) is the set of all objects that conform to type T. Namely:
  - *domain*(Integer) = set of all integers,
    domain(String) = set of all strings, etc.
  - *domain*(T), where T is reference type is the extent of T, ie, oids of all objects in class T
  - *domain*($[A_1: T_1, \ldots, A_n: T_n]$) is the set of all tuple values of the form $[A_1: v_1, \ldots, A_n: v_n]$, where each $v_i \in domain(T_i)$
  - *domain*($\{T\}$) is the set of all finite sets of the form $\{w_1, \ldots, w_m\}$, where each $w_i \in domain(T)$

27

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.)

28

Database Instance

- <u>*Set of extents*</u> for each class in the schema
- Each *object in the extent of a class must have the type of that class*, i.e., it must belong to the domain of the type
- Each object in the database must have *unique oid*
- The extents *must satisfy the constraints* of the database schema

29

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

30

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* = $[Attr_1: v_1, \ldots, Attr_n: v_n]$
- *multiset-value* = $\{v_1, \ldots, v_n\}$

31

SQL:1999 Tuple Values

- *Tuple value*: $[Attr_1: v_1, \ldots, Attr_n: v_n]$
  - $Attr_i$ are all distinct attributes
  - Each v_i is one of these:
    - Primitive value: a constant of type CHAR(…), INTEGER, FLOAT, etc.
    - Reference value: an object Id
    - Another tuple value
    - A collection value

      MULTISET introduced in SQL:2003.

      ARRAY– a fixed size array

32

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
  - 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))
)
```

33

Row Types (Contd.)

- Use path expressions to refer to the components of row types:

```
SELECT  P.Name
FROM    PERSON P
WHERE  P.Address.ZIP = '11794'
```

- Update operations:

```
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'
```

34

User Defined Types (UDT)

- UDTs allow specification of complex objects/tupes, 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

35

A UDT Example

```
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';
```

File that holds the binary code

36

Using UDTs in CREATE TABLE

- As an *attribute* type:

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

A previously defined UDT

- As a *table* type:

```
CREATE TABLE STUDENT OF StudentType;
```

Such a table is called *typed table.*

37

Objects

- Only typed tables contain objects (ie, 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 <u>exactly the same structure</u>
- Only the tuples in STUDENT – not STUDENT1 – have oids
- Will see later how to reference objects, create them, etc.

38

Querying UDTs

- Nothing special – just use path expressions

```
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.

39

Updating User-Defined Types

- Inserting a record into TRANSCRIPT:

```
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***, ie, 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

40

Observer Methods

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

```
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

41

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* → *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 StudentType:
  - *Id*: INTEGER → StudentType
  - *Name*: CHAR(20) → StudentType
  - *Address*: ROW(INTEGER, CHAR(20), CHAR(5)) → StudentType

42

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'
)

Add a value for *Id*

Create a blank *StudentType* object

Add a value for *Status*

Add a value for the *Address* attribute

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

43

Example: Changing a UDT Value

UPDATE TRANSCRIPT
SET *Student* = *Student*.*Address*(ROW(21,'Maple St.','12345')) .*Name*('John Smith'),
Grade = 'B'

Change *Address*

Change *Name*

WHERE *Student.Id* = 111111111 AND *CrsCode* = 'CS532' AND *Semester* = 'S2002'

- Mutators are used to change the values of the attributes *Address* and *Name*

44

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 *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

45

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 StudentType
REF IS stud_oid;
```

Self-referencing column

Self-referencing columns can be used in queries just like regular columns
Their *values cannot be changed*, however

46

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)
)
```

Reference type

Typed table where the values are drawn from

- 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!

47

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 " →" instead of "."

```
SELECT T.Student→Name, T.Grade

FROM TRANSCRIPT1 T
WHERE
    T.Student→Address.Street = "Main St."
```

Not crossing REF(..) boundary, use "."

Crossing REF(..) boundary, use →

48

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'
```

Explicit self-referential column of STUDENT2

49

Collection Data Types

- Set (multiset) data type was added in SQL:2003.

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

A bunch of references to objects of type CourseType

50

Querying Collection Types

- For each student, list the Id, address, and the courses in which the student is enrolled (assume STUDENT is a table of type StudentType):

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

Convert multiset to table

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

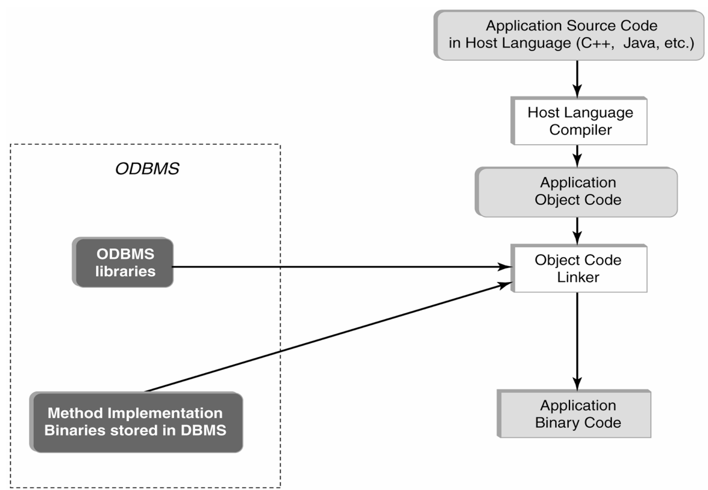
51

The ODMG Standard

- ODMG 3.0 was released in 2000
- Includes the data model (more or less)
- *ODL*: The object definition language
- *OQL*: The object query language
- A transaction specification mechanism
- *Language bindings*: How to access an ODMG database from C++, Smalltalk, and Java (expect C# to be added to the mix)

52

The Structure of an ODMG Application



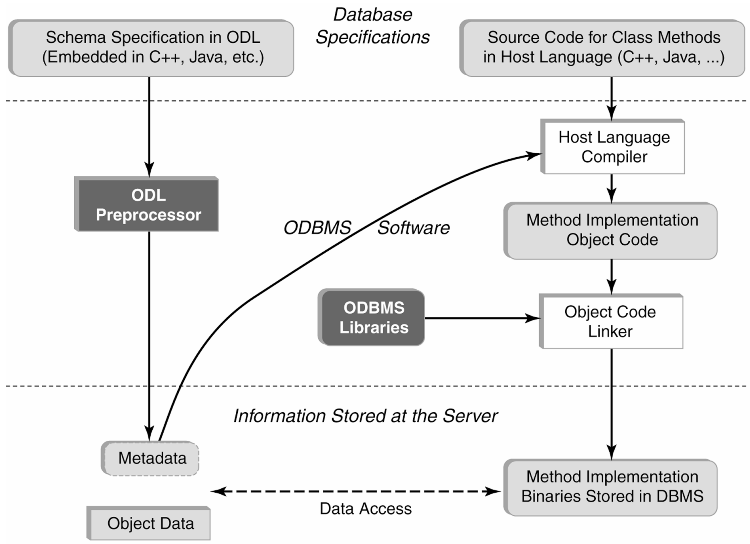
53

Main Idea: *Host Language = Data Language*

- Objects in the host language are mapped directly to database objects
- Some objects in the host program are ***persistent***. Think of them as "proxies" of the actual database objects. Changing such objects (through an assignment to an instance variable or with a method application) directly and transparently affects the corresponding database object
- Accessing an object using its oid causes an "***object fault***" similar to pagefaults in operating systems. This transparently brings the object into the memory and the program works with it as if it were a regular object defined, for example, in the host Java program

54

Architecture of an ODMG DBMS



55

SQL Databases vs. ODMG

- *In SQL*: Host program accesses the database by sending SQL queries to it (using JDBC, ODBC, Embedded SQL, etc.)
- *In ODMG*: Host program works with database objects directly
- ODMG has the facility to send OQL queries to the database, but this is viewed as evil: brings back the impedance mismatch

56

ODL: ODMG's Object Definition Language

- Is rarely used, if at all!
  - *Relational databases*: SQL is the only way to describe data to the DB
  - *ODMG databases*: can do this directly in the host language
  - Why bother to develop ODL then?
- Problem: Making database objects created by applications written in different languages (C++, Java, Smalltalk) *interoperable*
  - Object modeling capabilities of C++, Java, Smalltalk are very different.
  - How can a Java application access database objects created with C++?
- *Hence*: Need a *reference data model*, a common target to which to map the language bindings of the different host languages
  - *ODMG says*: Applications in language A can access objects created by applications in language B if these objects map into a subset of ODL supported by language A

57

ODMG Data Model

- Classes + inheritance hierarchy + types
- Two kinds of classes: "*ODMG classes*" and "*ODMG interfaces*", similarly to Java
  - An ODMG interface:
    - has no method code – only signatures
    - does not have its own objects – only the objects that belong to the interface's ODMG subclasses
    - cannot inherit from (be a subclass of) an ODMG class – only from another ODMG interface (in fact, from multiple such interfaces)
  - An ODMG class:
    - can have methods with code, own objects
    - can inherit from (be a subclass of) other ODMG classes or interfaces
      - can have at most one immediate superclass (but multiple immediate super-interfaces)

58

ODMG Data Model (Cont.)

- Distinguishes between objects and pure values (values are called *literals*)
  - Both can have complex internal structure, but only objects have oids

59

Example

```
interface  PersonInterface: Object  {        // Object is the ODMG topmost interface
     attribute String Name;
     attribute String SSN;
     Integer  Age();
}
class PERSON: PersonInterface                // inherits from ODMG interface
     (  extent PersonExt                     // note:  extents have names
        keys SSN, (Name, PhoneN)   ) :  persistent;
{  attribute ADDRESS Address;
   attribute Set<String> PhoneN;
   attribute enum SexType {m,f} Sex;
   attribute  date DateOfBirth;
   relationship  PERSON  Spouse;                  // note:  relationship vs. attribute
   relationship  Set<PERSON> Child;
   void  add_phone_number(in String  phone);    // method signature
}
struct ADDRESS {  // a literal type (for pure values)
     String StNumber;
     String StName;
}
```

60

More on the ODMG Data Model

- Can specify keys (also foreign keys – later)
- Class extents have their own names – this is what is used in queries
  - As if relation instances had their own names, distinct from the corresponding tables
- Distinguishes between *relationships* and *attributes*
  - Attribute values are literals
  - Relationship values are objects
  - ODMG relationships have little to do with relationships in the E-R model – do not confuse them!!

61

Example (contd.)

```
class STUDENT extends PERSON {
    ( extent StudentExt )
     attribute  Set<String> Major;
     relationship Set<COURSE> Enrolled;
}
```

- STUDENT is a subclass of PERSON (both are classes, unlike ADDRESS in the previous example)
- A class can have at most one immediate superclass
- No *name overloading*: a method with a given name and signature cannot be inherited from more than one place (a superclass *or* super-interface)

62

Referential Integrity

```
class STUDENT extends PERSON {
    ( extent StudentExt )
    attribute Set<String> Major;
    relationship Set<COURSE> Enrolled;
}
class COURSE: Object {
    ( extent CourseExt )
    attribute Integer CrsCode;
    attribute String Department;
    relationship Set<STUDENT> Enrollment;
}
```

- *Referential integrity*: If JoePublic takes CS532, and CS532 $\in$ JoePublic.*Enrolled*, then deleting the object for CS532 will delete it from the set JoePublic.*Enrolled*
- Still, the following is possible:

  CS532 $\in$ JoePublic.*Enrolled* *but* JoePublic $\notin$ CS532.*Enrollment*
- **Question**: Can the DBMS automatically maintain consistency between JoePublic.*Enrolled* and CS532.*Enrollment*?

63

Referential Integrity (Contd.)

Solution:

```
class STUDENT extends PERSON {
    ( extent StudentExt )
    attribute Set<String> Major;
    relationship Set<COURSE> Enrolled
                 inverse COURSE::Enrollment;
}
class COURSE: Object {
    ( extent CourseExt )
    attribute Integer CrsCode;
    attribute String Department;
    relationship Set<STUDENT> Enrollment
                 inverse STUDENT::Enrolled;
}
```

64

OQL: The ODMG Query Language

- Declarative
- SQL-like, but better
- Can be used in the *interactive* mode
  - Very few vendors support interactive use
- Can be used as *embedded* language in a host language
  - This is how it is usually used
  - OQL brings back the impedance mismatch

65

Example: Simple OQL Query

SELECT DISTINCT S.*Address*
FROM PersonExt S
WHERE S.*Name* = 'Smith"

- Can hardly tell if this is OQL or SQL
- Note: Uses the name of the extent of class PERSON, *not* the name of the class

66

Example: A Query with Method Invocation

- Method in the SELECT clause:

SELECT M.*frameRange*(100, 1000)
FROM MOVIE M
WHERE M.*Name* = "The Simpsons"

- Method with a side effect:

SELECT S.*add_phone_number*("555-1212")
FROM PersonExt S
WHERE S.*SSN* = "123-45-6789"

67

OQL Path Expressions

- Path expressions can be used with attributes:

SELECT DISTINCT S.*Address*.*StName*
FROM PersonExt S
WHERE S.*Name* = "Smith"

Attribute

- As well as with relationships:

SELECT DISTINCT S.*Spouse*.*Name*
FROM PersonExt S
WHERE S.*Name* = "Smith"

Relationship

68

Path Expressions (Contd.)

- Must be *type consistent*: the type of each prefix of a path expression must be consistent with the method/attribute/relationship that follows
- For instance, is S is bound to a PERSON object, then S.*Address*.*StName* and S.*Spouse*.*Name* are type consistent:
    - PERSON objects have attribute Address and relationship Spouse
    - S.Address is a literal of type ADDRESS; it has an attribute StName
    - S.Spouse is an object of type PERSON; it has a attribute Name, which is inherited from PersonInterface

69

Path Expressions (Contd.)

- Is P.*Child*.*Child*.*PhoneN* type consistent (P is bound to a PERSON objects)?
    - In some query languages, but not in OQL!
- *Issue*: Is P.*Child* a single set-object or a set *of* objects?
    1. If it is a set of PERSON objects, we can apply *Child* to each such object and P.*Child*.*Child* makes sense (as a set of grandchild PERSON objects)
    2. If it is a single set-object of type Set<PERSON>, then P.*Child*.*Child* *does not* make sense, because such objects do not have the *Child* relationship
- *OQL uses the second option.* Can we still get the phone numbers of grandchildren? – Must *flatten* out the sets:

    flatten(flatten(P.*Child*).*Child*).*Phone*

    – A bad design decision. We will see in Chapter 17 that XML query languages use option 1.

70

Nested Queries

- As in SQL, nested OQL queries can occur in
  - The FROM clause, for virtual ranges of variables
  - The WHERE clause, for complex query conditions
- In OQL nested subqueries can occur in SELECT, too!
  - Do nested subqueries in SELECT make sense in SQL?

What does the next query do?

```
SELECT  struct{ name:    S.Name,
                courses: (SELECT  E
                          FROM    S.Enrolled  E
                          WHERE  E.Department="CS" )
               }
FROM  StudentExt  S
```

71

Aggregation and Grouping

- The usual aggregate functions *avg, sum, count, min, max*
- In general, do not need the GROUP BY operator, because we can use nested queries in the SELECT clause.
  - For example: *Find all students along with the number of Computer Science courses each student is enrolled in*

```
SELECT   name :  S.Name
         count:   count( SELECT  E.CrsCode
                         FROM    S.Enrolled  E
                         WHERE   E.Department = "CS"  )
FROM   StudentExt  S
```

72

Aggregation and Grouping (Contd.)

- GROUP BY/HAVING exists, but does not increase the expressive power (unlike SQL):

```
SELECT   S.Name,  count: count(E.CrsCode)
FROM  StudentExt  S,  S.Enrolled E
WHERE  E.Department = "CS"
GROUP BY  S.SSN
```

Same effect, but the optimizer can use it as a hint.

73

GROUP BY as an Optimizer Hint

```
SELECT
    name :  S.Name
    count:   count(SELECT  E.CrsCode
                   FROM    S.Enrolled  E
                   WHERE   E.Department = "CS"  )
FROM   StudentExt  S
```

The query optimizer would compute the inner query for each s∈ StudentExt, so s.*Enrolled* will be computed for each s.

If enrollment information is stored separately (not as part of the STUDENT Object), then given s, index is likely to be used to find the corresponding courses. Can be expensive, if the index is not clustered

```
SELECT   S.Name,  count: count(E.CrsCode)
FROM  StudentExt  S,  S.Enrolled E
WHERE  E.Department = "CS"
GROUP BY  S.SSN
```

The query optimizer can recognize that it needs to find all courses for each student. It can then sort the enrollment file on student oids (thereby grouping courses around students) and then compute the result in one scan of that sorted file.

74

ODMG Language Bindings

- A set of interfaces and class definitions that allow host programs to:
  - Map host language classes to database classes in ODL
  - Access objects in those database classes by *direct manipulation* of the mapped host language objects
- Querying
  - Some querying can be done by simply applying the methods supplied with the database classes
  - A more powerful method is to send OQL queries to the database using a statement-level interface (which makes impedance mismatch)

75

Java Bindings: Simple Example

public class STUDENT extends PERSON {
    public DSet *Major*;
    ……..
}

Cant say "set of strings" – a Java limitation

- DSet class
  - part of ODMG Java binding, extends Java Set class
  - defined because Java Set class cannot adequately replace ODL's Set<…>

STUDENT X;
… … …
X.*Major*.*add*("CS");
… … …

add( ) is a method of class DSet (a modified Java's method). If X is bound to a persistent STUDENT object, the above Java statement will change that object in the database

76

Language Bindings: Thorny Issues

- Host as a data manipulation language is a powerful idea, but:
  - Some ODMG/ODL facilities do not exist in some or all host languages
  - The result is the lack of syntactic and conceptual unity
- Some issues:
  - Specification of *persistence* (which objects persist, ie, are automatically stored in the database by the DBMS, and which are *transient*)
    - First, a class must be declared *persistence capable* (differently in different languages)
    - Second, to actually make an object of a persistence capable class persistent, different facilities are used:
      - In C++, a special form of new() is used
      - In Java, the method makePersistent() (defined in the ODMG-Java interface Database) is used
  - Representation of relationships
    - Java binding does not support them; C++ and Smalltalk bindings do
  - Representation of literals
    - Java & Smalltalk bindings do not support them; C++ does

77

Java Bindings: Extended Example

- The OQLQuery class:

```
class OQLQuery {
    public OQLQuery(String query);   // the query constructor
    public bind(Object parameter);   // explained later
    public Object execute();         // executes queries
    … … several more methods ……
}
```

- Constructor: *OQLQuery*('SELECT …')
  - Creates a query object
  - The query string can have *placeholders* $1, $2, etc., like the ` ?' placeholders in Dynamic SQL, JDBC, ODBC. (Why?)

78

Extended Example (Cont.)

- Courses taken exclusively by CS students in Spring 2002:

```
DSet  students,courses;
String semester;
OQLQuery  query1, query2;
query1 = new  OQLQuery("SELECT  S  FROM  STUDENT  S  "
                                  +  "WHERE  \"CS\"  IN  S.Major");
students = (DSet) query1.execute();
query2 = new OQLQuery("SELECT  T  FROM  COURSE  T  "
                                 +  "WHERE  T.Enrollment.subsetOf($1)  "
                                 +  "AND  T.Semester = $2");
semester = new String("S2002");
query2.bind(students);    // bind $1 to the value of the variable students
query2.bind(semester);   // bind $2 to the value of the variable semester
courses = (DSet) query2.execute();
```

79

Interface DCollection

- Allows queries (select) from collections of database objects
- DSet inherits from DCollection, so, for example, the methods of DCollection can be applied to variables courses, students (previous slide)

```
public interface  DCollection  extends java.util.Collection {
    public DCollection  query(String condition);
    public Object  selectElement(String condition);
    public Boolean  existsElement(String condition);
    public java.util.Iterator  select(String condition);
}
```

80

Extended Example (Cont.)

- query(*condition*) – selects a subcollection of objects that satisfy *condition*:

```
DSet seminars;
seminars = (DSet) courses.query(“this. Credits = 1”);
```

- select(*condition*) – like query( ), but creates an iterator; can now scan the selected subcollection object-by-object:

```
java.util.Iterator  seminarIter;
Course  seminar;
seminarIter = (java.util.Iterator) courses.select(“this. Credits=1”);
while ( seminar=seminarIter.next( ) ) {
    … … …
}
```

81

CORBA:
Common Object Request Broker Architecture

- Distributed environment for clients to access objects on various servers
- Provides *location transparency* for distributed computational resources
- Analogous to *remote procedure call* (RPC) and *remote method invocation* in Java (RMI) in that all three can invoke remote code.
- But CORBA is more general and defines many more protocols (eg, for object persistence, querying, etc.). In fact, RMI is implemented using CORBA in Java 2

82

Interface Description Language (IDL)

- Specifies interfaces only (ie, classes without extents, attributes, etc.)
- No constraints or collection types

```
// File Library.idl
module Library {
    interface myLibrary{
        string searchByKeywords(in string keywords);
        string searchByAuthorTitle(in string author, in string title);
    }
}
```

83

Object Request Broker (ORB)

- Sits between clients and servers
- Identifies the actual server for each method call and dispatches the call to that server
- Objects can be implemented in different languages and reside on dissimilar OSs/machines, so ORB converts the calls according to the concrete language/OS/machine conventions

84

ORB Server Side

- Library.idl → IDL Compiler → *Library-stubs.c, Library-skeleton.c*

  → Method signatures to *interface repository*
- *Server skeleton*: Library-skeleton.c
  - Requests come to the server in OS/language/machine independent way
  - Server objects are implemented in some concrete language, deployed on a concrete OS and machine
  - Server skeleton *maps* OS/language/machine independent requests to calls understood by the concrete implementation of the objects on the server
- *Object adaptor*: How does ORB know which server can handle which method calls? – Object adaptor, a part of ORB
  - When a server starts, it *registers* itself with the ORB object adaptor
  - Tells which method calls in which interfaces it can handle. (Recall that method signature for all interfaces are recorded in the interface repository).
- *Implementation repository*: remembers which server implements which methods/interfaces (the object adaptor stores this info when a server registers)

85

ORB Client Side

- *Static invocation*: used when the application knows which exactly method/interface it needs to call to get the needed service
- *Dynamic invocation*: an application might need to figure out what method to call by querying the interface repository
  - For instance, an application that searches community libraries, where each library provides different methods for searching with different capabilities. For instance, some might allow search by title/author, while others by keywords. Method names, argument semantics, even the number of arguments might be different in each case

86

Static Invocation

- *Client stub*: Library-stubs.c
  - For static invocation only, when the method/interface to call is known
  - Converts OS/language/machine specific client's method call into the OS/language/machine independent format in which the request is delivered over the network
    - This conversion is called *marshalling of arguments*
    - Needed because client and server can be deployed on different OS/machine/etc.
    - Consider: 32-bit machines vs. 64 bit, little-endian vs. big endian, different representation for data structures (eg, strings)
  - Recall: the machine-independent request is unmarshalled on the server side by the server skeleton
  - Conversion is done *transparently* for the programmer – the programmer simply links the stub with the client program

87

Dynamic Invocation

- Used when the exact method calls are not known
- Example: Library search service
  - Several community libraries provide CORBA objects for searching their book holdings
  - New libraries can join (or be temporarily or permanently down)
  - Each library has its own legacy system, which is wrapped in CORBA objects. While the wrappers might follow the same conventions, the search capabilities of different libraries might be different (eg, by keywords, by wildcards, by title, by author, by a combination thereof)
  - User fills out a Web form, unaware of what kind of search the different libraries support
  - The user-side search application should
    - take advantage of newly joined libraries, even with different search capabilities
    - continue to function even if some library servers are down

88

Dynamic Invocation (Contd.)

- Example: IDL module with different search capabilities

```
module Library {
    interface  library1 {
        string  searchByKeywords(in string keywords);
        string  searchByAuthorTitle(in string author, in string title);
    }
    interface  library2 {
        void  searchByTitle(in string title, out string result);
        void  searchByWildcard(in string wildcard, out string result);
    }
}
```

The client application:

- Examines the fields in the form filled out by the user
- **Examines the interface repository** – next slide
- Decides which methods it can call with which arguments
- **Constructs the actual call** – next slide

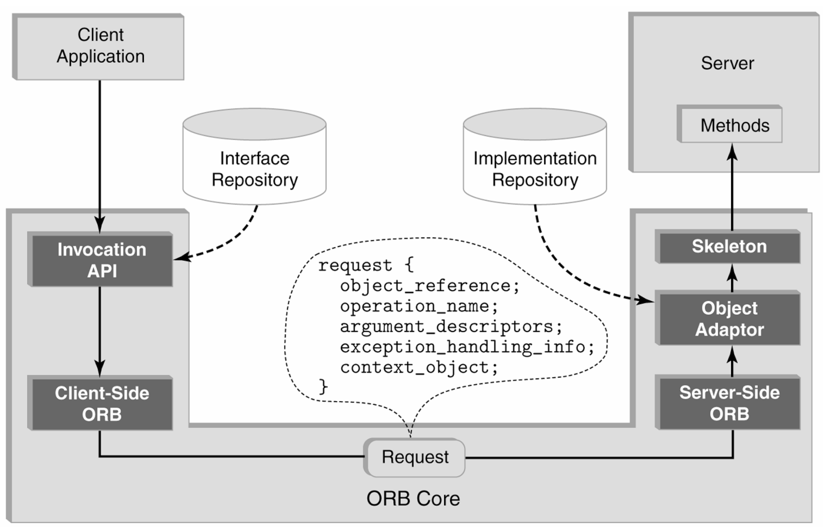
89

Dynamic Invocation API

- Provides methods to query the *interface repository*
- Provides methods to construct machine-independent requests to be passed along to the server by the ORB
- Once the application knows which method/interface to call with which arguments, it constructs a ***request***, which includes:
  - Object reference (which object to invoke)
  - Operation name (which method in which interface to call)
  - Argument descriptors (argument names, types, values)
  - Exception handling info
  - Additional "context" info, which is not part of the method argum ents
- *Note*: The client stub is essentially a piece of code that uses the dynamic invocation API to create the above requests. Thus:
  - With *static invocation*, the stub is created *automatically* by the IDL compiler
  - With *dynamic invocation*, the programmer has to *manually* write the code to create and invoke the requests, because the requisite information is not available at compile time

90

CORBA Architecture

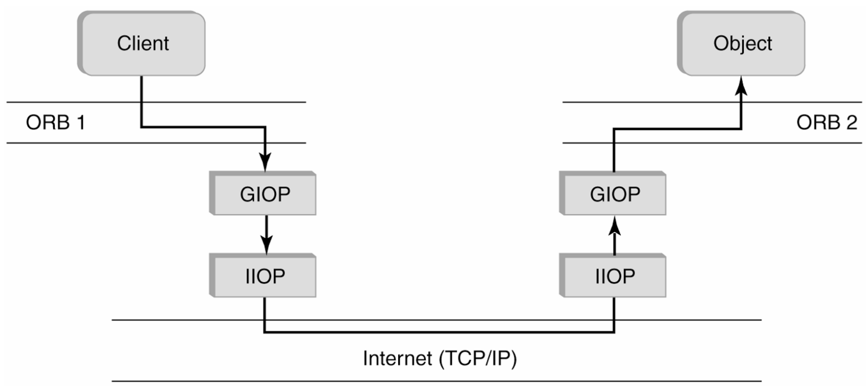


Interoperability within CORBA

- ORB allows objects to talk to each other if they are registered with that ORB; can objects registered with different ORBs talk to each other?
- *General inter-ORB protocol (GIOP)*: message format for requesting services from objects that live under the control of a different ORB
  - Often implemented using TCP/IP
  - Internet inter-ORB protocol (IIOP) specifies how GIOP messages are encoded for delivery via TCP/IP

92

Inter-ORB Architecture



93

CORBA Services

- Rich infrastructure on top of basic CORBA
- Some services support database-like functions:
  - *Persistence services* – how to store CORBA objects in a database or some other data store
  - *Object query services* – how to query persistent CORBA objects
  - *Transaction services* – how to make CORBA applications atomic (either execute them to the end or undo all changes)
  - *Concurrency control services* – how to request/release locks. In this way, applications can implement transaction isolation policies, such as two-phase commit

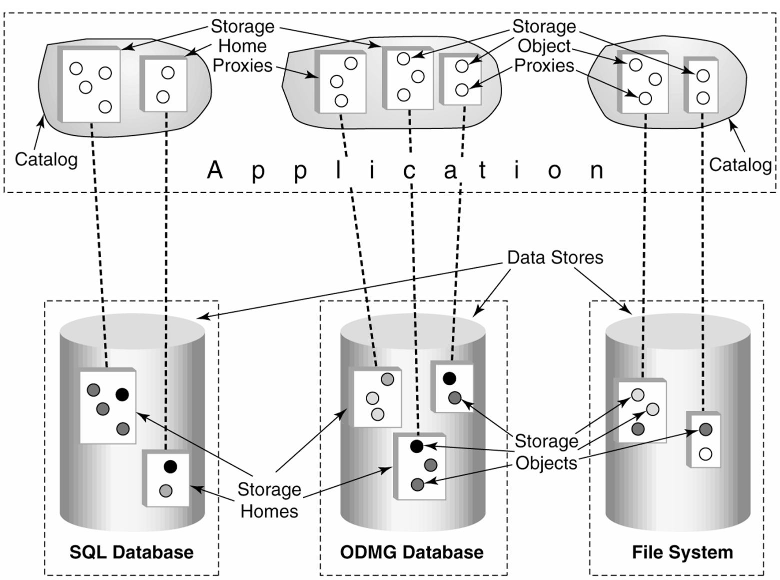
94

Persistent State Services (PSS)

- PSS – a standard way for data stores (eg, databases, file systems) to define interfaces that can be used by CORBA clients to manipulate the objects in that data store
- On the server:
  - Objects are in *storage homes* (eg, classes)
  - Storage homes are grouped in *data stores* (eg, databases)
- On the client:
  - Persistent objects (from the data store) are represented using *storage object proxies*
  - Storage object proxies are organized into *storage home proxies*
- Clients manipulate storage object proxies *directly*, like ODMG applications do

95

CORBA Persistent State Services

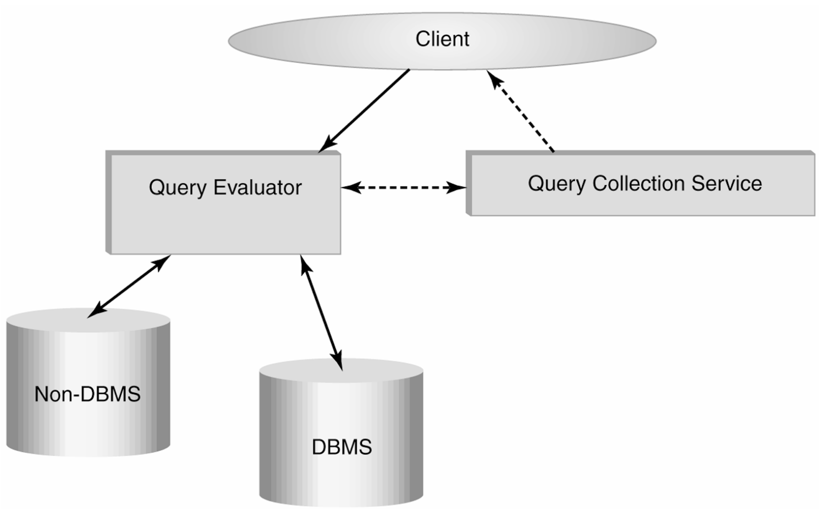


Object Query Services (OQS)

- OQS makes it possible to query persistent CORBA objects
- Supports SQL and OQL
- Does two things:
  - *Query evaluator*: Takes a query (from the client) and translated it into the query appropriate for the data store at hand (eg, a file system does not support SQL, so the query evaluator might have quite some work to do)
  - *Query collection service*: Processes the query result.
    - Creates an object of type *collection*, which contain references to the objects in the query result
    - Provides an *iterator object* to let the application to process each object in the result one by one

97

Object Query Services



98

Transaction and Concurrency Services

- *Transactional services*:
  - Allow threads to become transactions. Provide
    - *begin*()
    - *rollback*()
    - *commit*()
  - Implement *two-phase commit protocol* to ensure atomicity of distributed transactions
- *Concurrency control services*:
  - Allow transactional threads to request and release locks
  - Implement *two-phase locking*
  - Only supports – <u>does not enforce</u> – isolation. Other, non-transactional CORBA applications can violate serializability

99