Object Database Language
ODL – OQL

Object Definition Language – ODL

- ODL supports semantics constructs of ODMG.
- ODL is independent of any programming language.
- ODL is used to create object specification (classes and interfaces).
- ODL is not used for database manipulation.

ODL – OQL

- Standards group: ODMG = Object Data Management Group.
- ODL = Object Description Language, like CREATE TABLE part of SQL.
- OQL = Object Query Language, tries to imitate SQL in an OO framework.

Framework

- Assumption: OO-DBMS vendors implementing an OO language like C++ with extensions (OQL) that allow the programmer to transfer data between the database and “host language” seamlessly.
- ODL is used to define *persistent* classes – whose objects may be stored permanently in the database.
- ODL classes look like Entity sets with binary relationships, plus methods.
- ODL class definitions are part of the extended, OO host language.

ODL – Overview

- A class declaration includes:
  1. A name for the class.
  2. Optional *key* declaration(s).
  3. *Extent* declaration = name for the set of currently existing objects of the class.
  4. Element declarations. An *element* is either an attribute, a relationship, or a method.

ODL – Overview

- class <name>
  (extent …… key …..)
  {
      attribute 1 ;
      attribute 2 ;
      ……
      relationship ….. ;
      ….
      methods;
  };

ODL Types

- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors:
  - Struct for structures.
  - *Collection types* : Set, Bag, List, Array, and Dictionary.
- Relationship types can only be a class or a single collection type applied to a class.

ODL Keys

- You can declare any number of keys for a class.
- After the class name, add: (key <list of keys>)
- A key consisting of more than one attribute needs additional parentheses around those attributes.
- Example :
  - class Person (key ssn) { …

    ssn is the key for Person.
  - class Course (key (dept,number),(room, hours)) { ….

    dept and number form one key; so do room and hours.

ODL Extents

- For each class there is an *extent*, the set of existing objects of that class.
- Indicate the extent after the class name, along with keys, as:

  (extent <extent name> …)
- Example :
  - class Course

    (extent courses key name) { …
  - Conventionally, use singular for class names, plural for the corresponding extent.

ODL Extents

- Extents – used to distinguish *class definition* from *the set of objects of that class* that exist at a given time.
- Same as that between a *relation schema* and a *relation instance*.
- The class name is a schema for the class, while the extent is the name of the current set of objects of that class.
- The query language *OQL refers to the extent*, not to the class, when we want to examine the data currently stored in database.

Attribute Declarations

- Attributes are (usually) elements with a type that does not involve classes.

```
attribute <type> <name>;
```

Example:

```
class Degree {
        attribute string college;
        attribute string degree;
        attribute string year;
    };
```

Attribute Declarations

- Attributes can have a structure (as in C) or be an enumeration.
- Declare with

  attribute [Struct or Enum] <name of struct or enum>

  { <details> } <name of attribute>;

- Details are field names and types for a Struct, a list of constants for an Enum.

Attribute Declarations

- Example:

class Person {

attribute **struct** $\boxed{\text{Pname}}$ **{string** fname**, string** lname **}** $\boxed{\text{name;}}$

attribute string ssn;

attribute **enum** $\boxed{\text{Gender}}$ {M,F} $\boxed{\text{sex;}}$

};

Names for the structure and enumeration

names of the attributes

Relationship Declarations

- Relationships connect an object to one or more other objects of one class.

  relationship <type> <name>

  inverse <relationship>;

- Suppose class *C* has a relationship *R* to class *D*.
- Then class *D* must have some relationship *S* to class *C*.
- *R* and *S* must be true **inverses**:
  - If object *c* is related to object *d* by *R*, then *d* must be related to *c* by *S*.

Relationship Types

- The type of a relationship is either:
  - A class, like Faculty. If so, an object with this relationship can be connected to only one Faculty object.
  - Set<Faculty>: the object is connected to a set of Faculty objects.
  - Bag<Faculty>, List<Faculty>, Array<Faculty>: the object is connected to a bag, list, or array of Faculty objects.

Relationship Example

class Department {

attribute string dname;

attribute string dphone;

relationship set<Course> offers **inverse** Course::offered_by;

};

The type of relationship *offers* is a set of Course objects.

Relationship Cardinality

- All ODL relationships are binary.
- 1. *Many-many* relationship between class C and D:

  the type of the relationship is `Set<…>` at both sides
- 2. *Many-one* relationship from C to D:

  the type of the relationship in C (many-side) is D (*class*), while

  the type in D (one-side) is `Set<…>`
- 3. *One-many* relationship from C to D: reverse of the above (2)
- 4. For *one-one* relationship, the type of relationship is *class* at

  both sides.

  Note: the Set<...> could be replaced by another collection types such as list or bag

Relationship Cardinality – M:N

- **class** Department {
  attribute string dname;
  attribute string dphone;
  relationship set<Course> offers **inverse** Course::offered_by;
  };

  class Course {
  attribute string cno;
  attribute string cname;
  relationship set<Department> offered_by **inverse** Department::offers;
  };

Many-many uses Set<…> in both directions.

Relationship Cardinality – N:1

- **class** Department {

  attribute string dname;

  attribute string dphone;

  relationship set<Student> has_major **inverse** Student::major_in;

  };

  class Student {

  attribute string class;

  relationship Department major_in **inverse** Department::has_major;

  };

Many-one uses Set<…> only with the “one.”

Subclasses in ODL

- If C is a subclass of another class D, then

  declaration of class C is followed with keyword `extends`

  and the name D
- A subclass *inherits* all the properties of its superclass

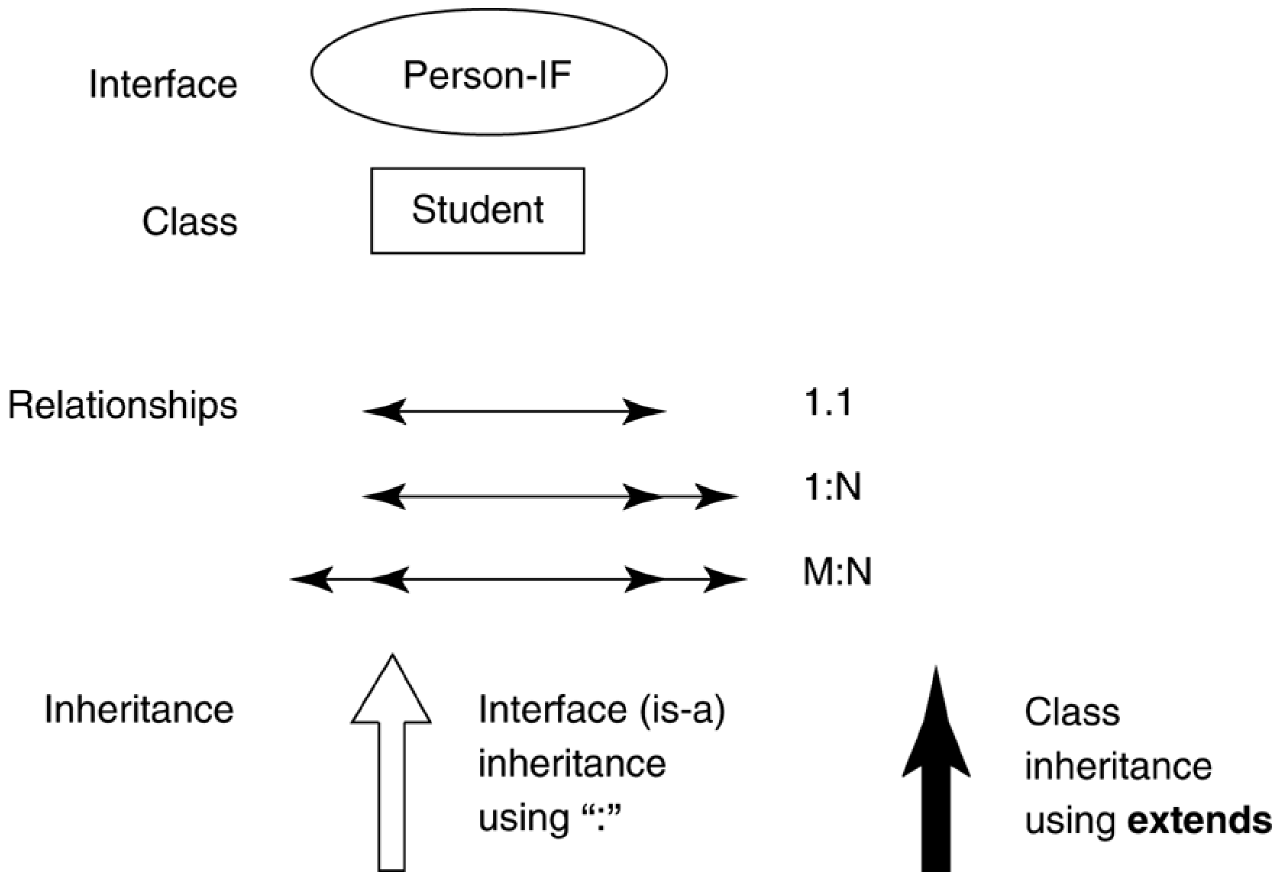
Subclasses in ODL

- **class** Person
  (extent persons, key aadhaar_no)
  {
  attribute string pname;
  attribute string aadhaar_no;
  attribute string birthdate;
  attribute string address;
  };

  Defines a subclass of superclass

  class Employee **extends** Person
  (extent employees)
  {
  attribute string designation;
  attribute string salary;
  attribute string dept;
  };

ODL Schema – graphical notation



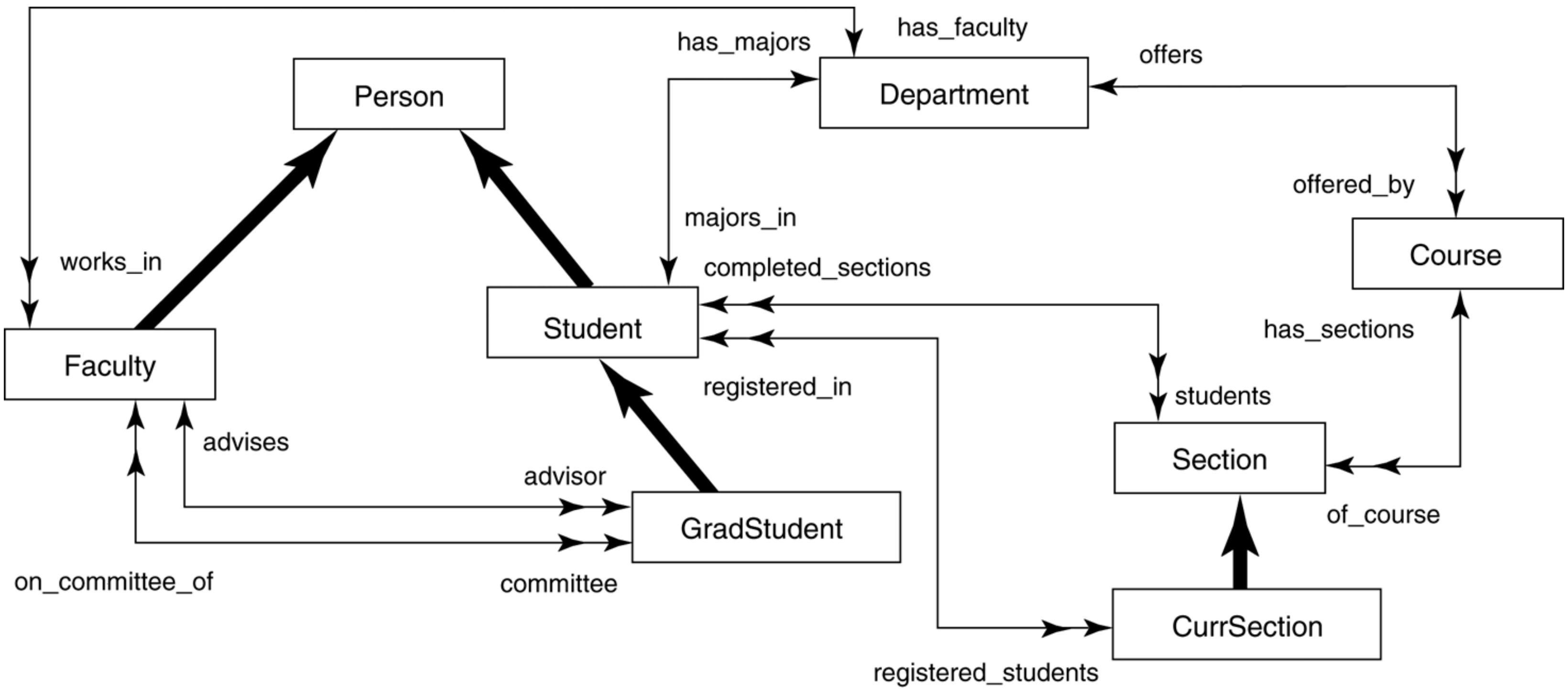
Method declarations

- A class definition may include declarations of methods for the class.
- Information consists of:
  1. Return type, if any.
  2. Method name.
  3. Argument modes and types (no names).

     Modes are in, out, and inout.
  4. Any exceptions the method may raise.

  Ex: `real gpa(in string) raises(noGrades);`

ODL Schema – University database



ODL ase

```
class Person
(   extent  persons
    key     ssn   )
{
    attribute    struct Pname {string fname, string mname, string mname }
                                          name;
    attribute    string                   ssn;
    attribute    date                     birthdate;
    attribute    enum Gender{M, F}        sex;
    attribute    struct Address {short no, string street, short aptno,
string city, string state, short zip }
                                          address;
    short    age();
};

class Faculty extends Person
(   extent  faculty   )
{
    attribute    string              rank;
    attribute    float               salary;
    attribute    string              office;
    attribute    string              phone;
    relationship Department          works_in inverse Department::has_faculty;
    relationship set<GradStudent> advises inverse GradStudent::advisor;
    relationship set<GradStudent> on_committee_of
                                     inverse GradStudent::committee;
    void    give_raise(in float raise);
    void    promote(in string new_rank);
};
```

ODL Schema – University database

```
class Grade
(    extent grades   )
{
     attribute       enum GradeValues{A,B,C,D,F,I,P}
                                             grade;
     relationship Section section inverse Section::students;
     relationship Student student inverse Student::completed_sections;
};

class Student extends Person
(    extent students   )
{
     attribute       string                  class;
     attribute       Department        minors_in;
     relationship Department majors_in inverse Department::has_majors;
     relationship set<Grade> completed_sections inverse Grade::student;
     relationship set<CurrSection> registered_in
                                             inverse CurrSection::registered_students;
     void     change_major(in string dname) raises(dname_not_valid);
     float    gpa();
     void     register(in short secno) raises(section_not_valid);
     void     assign_grade(in short secno; in GradeValue grade)
                                             raises(section_not_valid,grade_not_valid);
};
```

ODL S ase

```
class Degree
{
    attribute       string              college;
    attribute       string              degree;
    attribute       string              year;
};

class GradStudent extends Student
(   extent  grad_students   )
{
    attribute       set<Degree>         degrees;
    relationship Faculty advisor inverse Faculty::advises;
    relationship set<Faculty> committee inverse Faculty::on_committee_of;
    void    assign_advisor(in string lname; in string fname)
                                raises(faculty_not_valid);
    void    assign_committee_member(in string lname; in string fname)
                                raises(faculty_not_valid);
};

class Department
(   extent  departments   )
{
    attribute       string              dname;
    attribute       string              dphone;
    attribute       string              doffice;
    attribute       string              college;
    attribute       Faculty             chair;
    relationship set<Faculty> has_faculty inverse Faculty::works_in;
    relationship set<Student> has_majors inverse Student::majors_in;
    relationship set<Course>  offers inverse Course::offered_by;
};
```

ODL Schema – University database

```
class Course
(    extent  courses   )
{
     attribute      string              cname;
     attribute      string              cno;
     attribute      string              description;
     relationship set<Section> has_sections inverse Section::of_course;
     relationship set<Department> offered_by inverse Department::offers;
};

class Section
(    extent  sections   )
{
     attribute      short               secno;
     attribute      string              year;
     attribute      enum Quarter{Fall, Winter, Spring, Summer}
                                        qtr;
     relationship set<Grade> students inverse Grade::section;
     relationship Course of_course inverse Course::has_sections;
};
class CurrSection extends Section
(    extent  current_sections       )
{
     relationship set<Student> registered.students inverse Student::registered_in
     void     register_student(in string ssn)
                   raises(student_not_valid, section_not_valid, section_full);
};
```

OQL – Object Query Language

OQL – Object Query Language

- Path expression
- Result type
- Collection
- Quantification
- Aggregation
- Grouping

OQL – Object Query Language

- Basic syntax: select…from…where…
- ```
  SELECT   d.name
  FROM     d in departments
  WHERE    d.college = 'Engineering';
  ```
- An **entry point** to the database is needed for each query
- An extent name (e.g., departments in the above example) may serve as an entry point.
  ```

Path expression

- A **path expression** is used to specify a path to attributes and objects in an entry point.
- A path expression starts at a persistent object name (or its iterator variable).
- The name will be followed by zero or more dot connected relationship or attribute names.

departments.Chair;

departments.Chair.Rank;

departments.Has_faculty; <--- relationship

Path expression

- Let x be an object of class C.
- If a is an attribute of C, then $x.a$ is the value of that attribute.
- If r is a relationship of C, then $x.r$ is the value to which x is connected by r.
  - Could be an object or a set of objects, depending on the type of r.
- If m is a method of C, then $x.m$ (…) is the result of applying m to x.

Select-From-Where

- We may compute relation-like collections by an OQL statement:

  SELECT <list of values>

  FROM <list of collections and names for typical members>

  WHERE <condition>

FROM clause

- Each term of the FROM clause is:

  <collection> <member name>

- A collection can be:
  - The extent of some class.
  - An expression that evaluates to a collection, e.g., certain path expressions like d.offers

Example – 1

- To retrieve the name and college for students minoring in a given department:

```
select  s.minors_in.college, s.name.fname
from  students  s
where  s.minors_in.dname = 'Business';
```

Students is the extent representing all Student objects; s represents each Student object, in turn.

Legal expressions. s.minors_in is a Department object.

Using Path Expression

- If a path expression denotes an object, you can extend it with another dot and a property of that object.
  - Example: s ; s.minors_in ; s.minors_in.dname
- If a path expression denotes a collection of objects, you cannot extend it, but you can use it in the FROM clause.
  - Example: s.registered_in

Return Data type

- The data type of a query result can be any type defined in the ODMG model.
- As a default, the type of the result of select-from-where is a Bag of Structs.
  - Struct has one field for each term in the SELECT clause.
- If SELECT has only one term, technically the result is a one-field struct.
  - But a one-field struct is identified with the element itself.

Example 2 – result type

- Retrieve the name and college of faculty from Computer Science department.

  select f.name.fname, d.college

  from departments d, d.has_faculty f

  where d.dname = “Computer Science”

- Has type:

  Bag(Struct(fname: string, college: string))

Example 3 – result type

- Add DISTINCT after SELECT to make the result type a set, and eliminate duplicates.

  select distinct f.name.fname, d.college

  from departments d, d.has_faculty f

  where d.dname = “Computer Science”

- Has type:

  Set(Struct(fname: string, college: string))

Collection

- Collections that are *lists or arrays* allow retrieving their **first**, **last**, and *i*th elements.
- OQL provides operators for ordering the results.
- Use an ORDER BY clause, to make the result a list of structs, ordered by whichever fields are in the ORDER BY clause.
  - Ascending (ASC) is the default; descending (DESC) is an option.
- Access list elements by index [1], [2],…
- Gives capability similar to SQL cursors.

Example 4 – Collection

- To retrieve the last name of the faculty member who earns the highest salary:

  first (**select** struct(faculty f.name.lname, salary f.salary)
  from f **in** faculty
  order by f.salary desc;)

- Has type:

  List(Struct(faculty: string, salary: float))

Example 5 – Collection

- To retrieve gpa of all senior students majoring in Computer Science:

```
gpaList = ( select    struct(s.name.lname, s.gpa)
            from      d in departments, s in d.has_majors
            where     d.dname='Computer Science' and s.class='senior'
            order by  gpa desc; )
```

- We can find the first () element on the list by gpaList[1], the next by gpaList[2], and so on.
- Example: the name of student with top gpa: top = gpaList[1].lname;

Example 6 – Collection

- OQL is *orthogonal* – attributes, relationships and operation names can be used interchangeably.

gpaList = (**select** struct(s.name.lname, s.gpa)
from s **in** students
where s.majors_in.dname='Computer Science' **and** s.class='senior'
order by gpa desc;)

Quantification

- OQL provides membership and quantification operators:
- Let
  - c is a collection expression,
  - b is boolean condition, and
  - e an element of the type of elements in collection c.

(e **in** c) is true if e is in the collection c.

(**for all** e **in** c: b) is true if *all e elements* of collection c satisfy b

(**exists** e **in** c: b) is true if *at least one e* in collection c satisfies b.

Example 7 – in

- Retrieve the names of students who has completed the course 'Database Systems – I'.

```
select   s.name.fname, s.name.lname
from     students s,
where    "Database Systems I" in
                ( select C.cname
                  from C in s.completed_sections.Section.of_course
                );
```

Example 8 – exists

- Find department in which, the faculty earns salary more than $50000.

select d.dname

from departments d,

where exists

g **in** d.has_faculty : g.salary > 50000 ;

At least one Faculty object for departments d has a salary above $50000.

Example 9 – exists

- List any graduate Computer Science major having 4.0 GPA.

  exists g **in**

  (select S

  from grad_students S

  where s.majors_in.dname = 'Computer Science' **)**

  : g.gpa = 4 ;

Example 10 – for all

- All the Computer Science graduate students must be advised by Computer Science faculty.

```
for all g in
    (select  S
    from     grad_students S
    where    S.majors_in.dname='Computer Science')
    : g.advisor in  (select  d.has_faculty
                     from    departments d
                     where   d.dname = 'Computer Science' )
```

Single element from Collections

- An OQL query returns a collection, such as a bag, set or list.
- OQL's *element* operator can be used to return a single element from a singleton collection that contains one element:

    element (**select** d

    from d **in** departments

    where d.dname = 'Software Engineering');

- If d is empty or has more than one elements, an **exception** is raised.

Dname is a key

Aggregation

- OQL supports a number of aggregate operators that can be applied to collections:

  `min, max, count, sum, avg`

- `Count` returns an integer; others return the same type as the collection type.

Example 11 – avg

- To compute the average GPA of all seniors majoring in Business:

```
avg (select  s.gpa
     from    s in students
     where   s.class = 'senior' and
                     s.majors_in.dname ='Business');
```

Example 12 – count

- To retrieve all department names that have more than 100 majors:

```
select   d.dname
from     d in departments
where    count (d.has_majors) > 100 ;
```

Example 13 – max

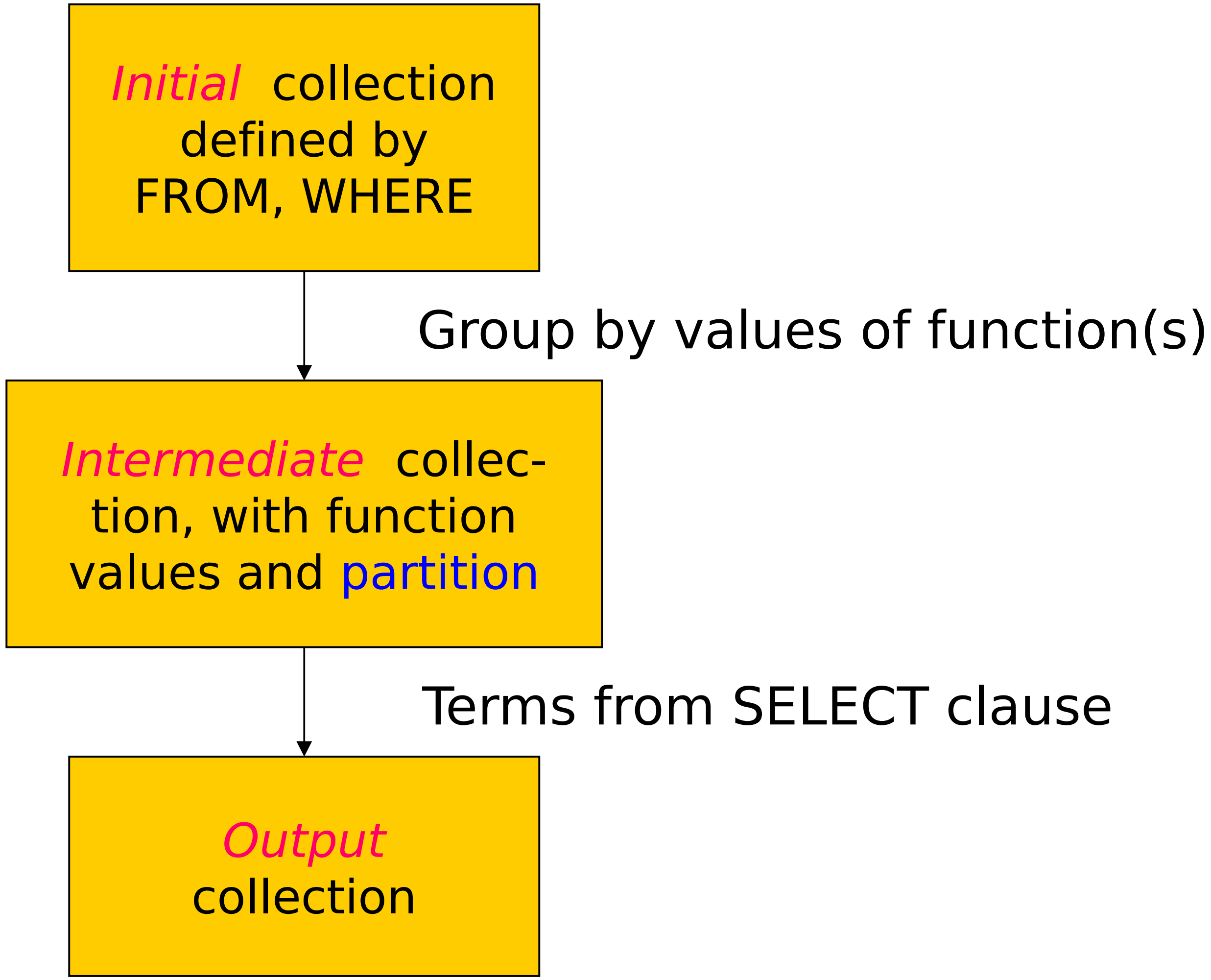
- To retrieve the highest salary earned by a faculty working in Computer Science department:

```
max (select s.salary
     from  s in faculty
     where s.works_in.dname = ‘Computer Science' );
```

Grouping

- OQL also supports a grouping operator called **group by.**
- OQL extends the grouping idea in several ways:
  - Any collection may be partitioned into groups.
  - Groups may be based on any function(s) of the objects in the initial collection.
  - Result of the query can be any function of the groups.

OQL Group by – outline



Intermediate collection

- OQL also supports a grouping operator called **group by.**
- Retrieve the number of majors in each department.

select struct(deptName, no_of_majors: count(partition))

from S **in** students

group by deptName: S.majors_in.dname;

One grouping function – Name is deptName,

type is string. Intermediate collection is a

set of structs with fields deptName: string and

partition: Set<Struct{S: students}>

Grouping

- Initial collection is based on FROM and WHERE – students S.
- The initial collection is a Bag of structs with one field for each "typical element" in the FROM clause.
- Here, a bag of structs of the form Struct(s: *obj*), where *obj* is a Students object.
- In general, bag of structs with one component for each function in the GROUP BY clause, plus one component always called *partition*.
- The partition value is the set of all objects in the initial collection that belong to the group represented by this struct.

Grouping

- A typical member of the intermediate collection in example is:

  Struct(deptName = "Computer Science", partition = $\{s_1, s_2, \ldots, s_n\}$)
- The output collection is computed by the SELECT clause, as usual.
- Without a GROUP BY clause, the SELECT clause gets the initial collection from which to produce its output.
- With GROUP BY, the SELECT clause is computed from the intermediate collection.

Example 14 – Group by, having clause

- Having clause can be used to filter the partitioned sets.
- To retrieve average GPA of majors in each department having >100 majors:

Average these gpa to create the value of field avg_gpa in the structs of the output collection.

select deptname,
avg_gpa: **avg** (**select** p.s.gpa **from** p **in** partition)
from s **in** students
group by deptname: s.majors_in.dname
having **count** (partition) > 100 ;

Initial collection: structs of the form Struct(s: Students object).

From each member *p* of the group's partition, get the field *s* (Students object), and from that object extract the gpa.

References

- Fundamentals of Database Systems, by Ramez Elamsri, Navathe.
- Lecture notes from Jeffrey Ullman.