

VU LEARNING HUB

vulearninghub.xyz

HIGHLIGHTED HANDOUT

CS713

Object Oriented DBMS

CS713 by VU Learning HUB

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

Object Oriented DBMS

Course Code: CS713

1	Introduction of pervious course	01
2	Postgres DBMS	15
3	Starburst DBMS	20
4	Open DB/ DDAFTER	26
5	Illustra DBMS	30
6	Object Oriented Data Model	34
7	Object Management Groups (ODMD)	60
8	C++ Bindings	88
9	OO support for Oracle	96
10	Objectivity/DB Objects	118
11	Programming Language Interface	127
12	OO Distributed Database Management	159

Presented by:

Dr. Nayyer Masood Dar

Ph.D Computer Science University of Bradford, UK

Lecture Videos:

<http://www.vumultan.com/CS713Video.aspx>

Referenced Books:

1. Database System: A Practical Approach to Design, Implementation and Management by T. Connolly, C. Begg
2. Object-Oriented Database Systems: Approaches and Architectures by C. Prabhu
3. The Object Data Standard: ODMG 3.0 by R. Cattell, D. Barry
4. Principles of Distributed Database Systems by T. Oszu, P. Valduriez, S. Sridhar
5. Object-oriented Oracle W. Rahayu, D. Taniar, E. Pardede

A World Class Education at Your Door Step

ورچوئل یونیورسٹی آف پاکستان

Lecture -1**Course Objectives**

- Why OO databases?
- Current state of Relational and Object-Relational Databases.
- Features of Object Oriented databases.
- Some leading OODBMSs.
- Research Issues in OO DBs.

References - 1

- T. Connolly, C. Begg, "Database System: A Practical Approach to Design, Implementation and Management", 4th Edition, Pearson Education.
- C. Prabhu, "Object-Oriented Database Systems: Approaches and Architectures", Third Edition, Prentice-Hall of India.
- R. Cattell, D. Barry, "The Object Data Standard: ODMG 3.0", Morgan Kaufman Publishers.
- T. Oszu, P. Valduriez, S. Sridhar, "Principles of Distributed Database Systems", Second Edition, ahayu, D. Taniar, E. Pardede, (editors) "Object-oriented OraclePearson Education.
- "Objectivity/DB Administration Manual", web site
<http://hepunx.rl.ac.uk/BRFOOT/www/Public/Computing/Databases/internal/Draft-administration1.pdf>
- W. Rahayu, D. Taniar, E. Pardede, (editors) "Object-oriented Oracle", IRM Press, 2006.

Course Prerequisites

- Database Management Systems
- Object Oriented Programming

Course of the Course-1

- The need of OO databases.
- Basics of Databases; as a revision.
- Features of Object-Orientation.
- Object-Oriented data models.
- Object-Relational DBMSs.
- Object-Oriented Database Management Systems (OODBMSs).
- Features of OODBMSs.
- Different OODBMSs.
- Research Issues in OO DBs.
- Course Revision.

Main Reference

- T. Connolly, C. Begg, "Database System: A Practical Approach to Design, Implementation and Management", Pearson Education, 4th Edition (Ch 25).

Main reasons for OODBs

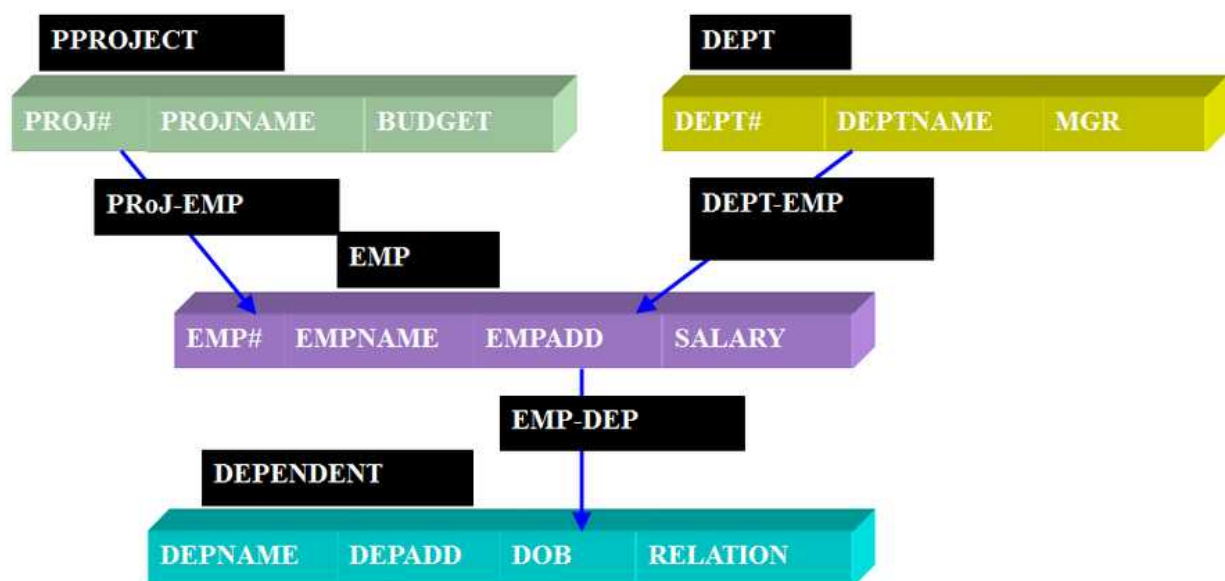
- Limitations of the Relational Data Model (currently dominated DM).
- Requirements of Advanced DB Applications.

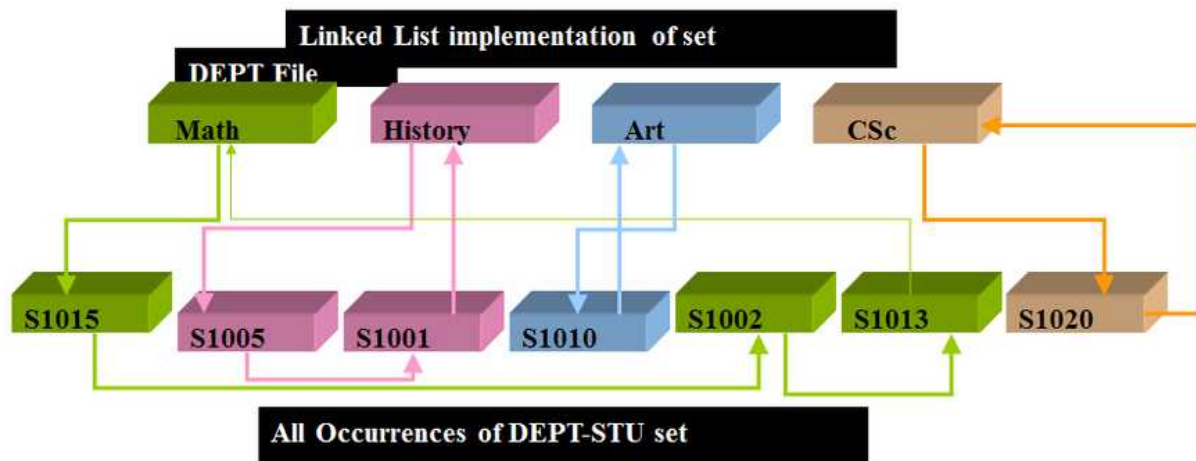
History of Data Models

- File-based approach for data processing.
- In early 60s research was initiated on a data handling system.
- Objective was to handle large amount of data efficiently.
- Mid-60s IMS developed.
- Arranges data in hierarchical form, in a tree-like structure.
- Same time network data model proposed.
- IDMS was the NDM based DBMS.
- CODASYL got involved and DBTG was established.

Network Data Model

- IN NDM, the database consists of a collection of set-type occurrences.
- Each set-type occurrence has one occurrence of OWNER RECORD, with zero or more occurrences of MEMBER RECORDS.
- To define a network database one needs to define:
 - ✓ The database record types which consist of data items.
 - ✓ The set-types.
- Uses network or plex structure as its basic data structure.
- Node corresponds to records types and link to the pointers.

Database Structure in NDM

Occurrence of Set Type in NDM**Problems faced in H & NDM**

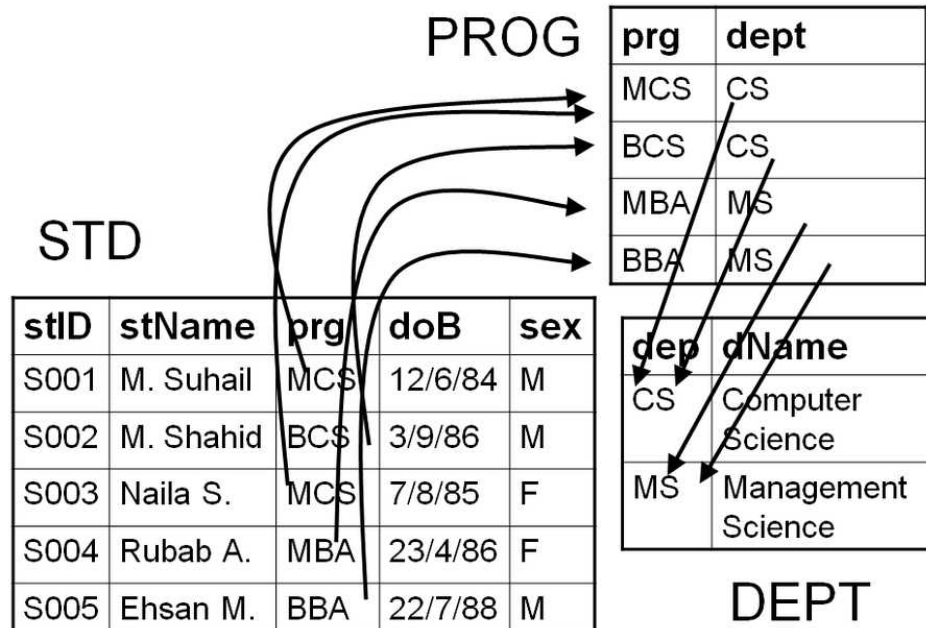
- Very complex structure from the application programmer's point of view.
- Difficult to design and use properly, because of the navigational nature of the data structure.
- Difficult to make changes in a database; no structural independence.
- No theoretical foundation.

Relational Data Model

- Proposed by E.F. Codd in 1970
- Major strengths
 - ✓ Mathematical foundation.
 - ✓ Higher degree of data independence.
 - ✓ Expanded set-oriented manipulation language.
- Components of RDM
 - ✓ Structure (Relation/Table).
 - ✓ Manipulation language (SQL).
 - ✓ Integrity Constraints (two).
- First two components have direct support from relational mathematics.
- So we have six basic properties of relational DB tables.

Six Basic Properties of Relational DB Tables or Relations

- Each cell of a table contains atomic/single value.
- Each column has a distinct name; the name of the attribute it represents.
- The values of the attributes come from the same domain.
- The order of the columns is immaterial.
- The order of the rows is immaterial.
- Each row/tuple/record is distinct, no two rows can be same.



Integrity Constraints

- Entity Integrity Constraint.
- Referential Integrity Constraint.
- Components of RDM
 - ✓ Structure (Relation/table).
 - ✓ Manipulation language (SQL).
 - ✓ Integrity Constraints.

BUTs with RDM

- Semantically weak; just one structure.
- Tables can store only atomic/single value.
- Normalized design recommended

Lecture -2

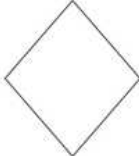
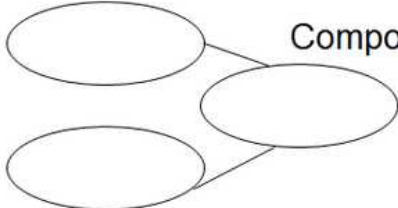
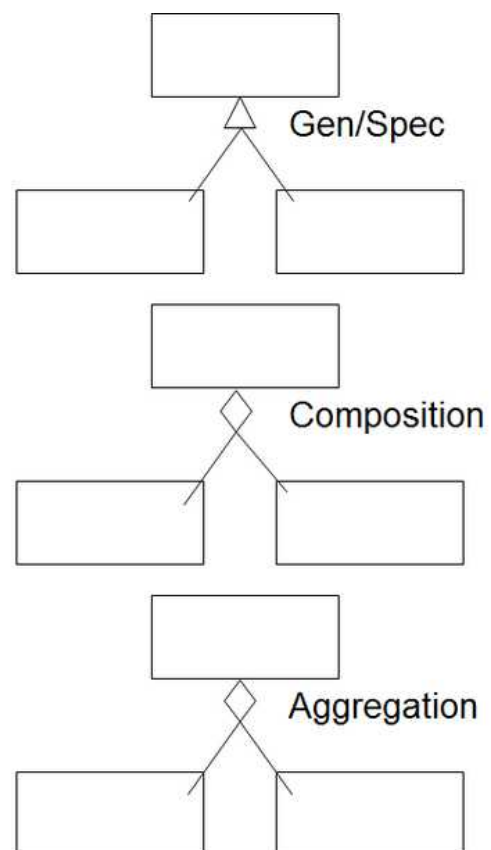
In comes ER DM

- Entity-Relationship data model defined in 1976 by Chen.
- Semantically rich data model.
- Facilitates in modeling by providing different structures to model different situations.

ER Data Model

- By constructs we mean:
- Entity Type
- Relationship
- Simple Attribute
- Composite Attribute
- Multi-valued Attribute
- Computed Attribute
- Enhanced ER DM provided even further constructs.

Symbols Used in ERDM

Entity Type Relationship Composite Attribute Computed Attribute Multi-valued Attribute 

Weaknesses of RDBMSs

- Poor Representation of “Real World” Entities.
 - ✓ Normalization leads to relations that do not correspond to entities in “real world”.
- Semantic Overloading
 - ✓ Relational model has only one construct for representing data and data relationships: the relation.
 - ✓ Relational model is semantically overloaded.
- Poor Support for Integrity and Enterprise Constraints.
- Homogeneous Data Structure
 - ✓ Relational model assumes both horizontal and vertical homogeneity.
 - ✓ Many RDBMSs now allow Binary Large Objects (BLOBs).
- Limited Operations
 - ✓ RDBMs only have a fixed set of operations which cannot be extended.
- Difficulty in Handling Recursive Queries
 - ✓ Recursive queries are supported but if only the depth of recursion is known.
 - ✓ Extension proposed to relational algebra to handle this type of query is unary transitive (recursive) closure operation.
- Impedance Mismatch
 - ✓ Most DMLs lack computational completeness.
 - ✓ To overcome this, SQL can be embedded in a high-level 3GL.
 - ✓ This produces an impedance mismatch - mixing different programming paradigms.
- Other Problems with RDBMSs
 - ✓ Transactions are generally short-lived and concurrency control protocols not suited for long-lived transactions.
 - ✓ Schema changes are difficult.
 - ✓ RDBMSs are poor at navigational access.

Main Reasons for OODBMS

- The limitations of RDM are debatable as there are database communities that do not agree or disagree at 100% with these points.
- The advanced DB Applications.

Advanced DB Applications**Computer-Aided Design (CAD)**

- Application generally related to engineering disciplines, like Mechanical, Civil or electrical etc.
- Relevant data may be about buildings, airplanes, and integrated circuit chips etc.
- Designs of this type have some common characteristics:
 - ✓ Data has many types, each with a small number of instances; contrary to a typical DB application.
 - ✓ Designs may be very large.
 - ✓ Design is not static but evolves through time; updates need to be propagated.

- ✓ Involves version control and configuration management.
- ✓ Updates are far-reaching.
- ✓ Cooperative engineering.

Computer-Aided Manufacturing (CAM)

- Stores similar data to CAD, plus data about discrete production.
- Applications must respond to real time events.
- Generally algorithms and custom rules are used to respond in a particular situation.

Computer-Aided Software Engineering (CASE)

- Stores data about stages of software development lifecycle.
- Design may be extremely large.
- Involves cooperative working.
- Need to maintain dependencies among components.
- Versioning and configuration management involved.

Network Management Systems

- Coordinate delivery of communication services across a computer network.
- Perform such tasks as network path management, problem management, and network planning.
- Office Information Systems (OIS) and Multimedia Systems.
- Digital Publishing.

Geographic Information Systems (GIS)

- Deal with spatial and temporal information, such as that used in land management and underwater exploration.
- May involve data from survey and satellite photographs; may be very large in size.

So we had the reasons

- Feature hungry applications.
- Limitations of the RDM.
- Popularity of OO paradigm.

Lecture - 3**Object-Oriented Concepts**

- Abstraction, encapsulation, information hiding.
- Objects and attributes.
- Object identity.
- Methods and messages.
- Classes, subclasses, superclasses, and inheritance.
- Overloading.
- Polymorphism and dynamic binding.

Abstraction

- **Process of identifying essential aspects of an entity and ignoring unimportant properties.** 
- Concentrate on what an object is and what it does, before deciding how to implement it.

Object

- An object is something uniquely identifiable, models a real-world entity and has got state and behavior.
 - ✓ Resembles entity, but object contains state and behavior and entity models only state.

Encapsulation & Information Hiding

- Encapsulation
 - ✓ Object contains both data structure and set of operations used to manipulate it.
- Information Hiding
 - ✓ Separate external aspects of an object from its internal details, which are hidden from outside.
- Allows internal details of an object to be changed without affecting applications that use it, provided external details remain same.
- Provides data independence.

Attributes

- Contain current state of an object
- Attributes can be classified as simple or complex.
 - ✓ Simple attribute can be a primitive type such as integer, string, etc., which takes on literal values.
 - ✓ Complex attribute can contain collections and/or references.
- Reference attribute represents relationship.
- **An object that contains one or more complex attributes is called a complex object.**

Object Identity

- Object IDentity (OID) associated with an object on its creation:
 - ✓ Generated by system.

- ✓ Unique to that object in the entire system.
- ✓ Used only by system, not by user.
- ✓ Independent of state of object
- ✓ Does not change.

Object Identity - Implementation

- In RDBMS OID is
 - ✓ Responsibility of the user.
 - ✓ Value based.
 - ✓ Unique only within table.
 - ✓ Key among attributes; so is state dependent.
 - ✓ Can be changed.
- Using disk or RAM address is also inefficient and may cause problems, like
 - ✓ They are small
 - ✓ On deletion memory can be reused that can cause problem.

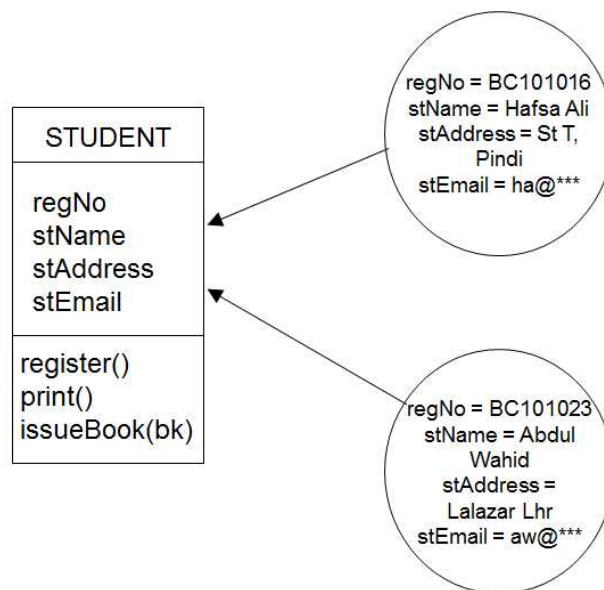
Methods and Messages

- Method
 - ✓ Implements functionality of object; is encapsulated
- Message
 - ✓ A request to an object to execute one of its methods.

Class

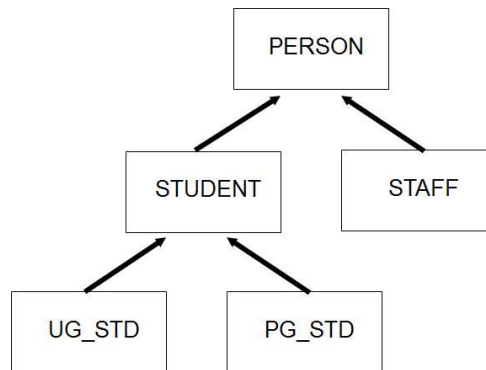
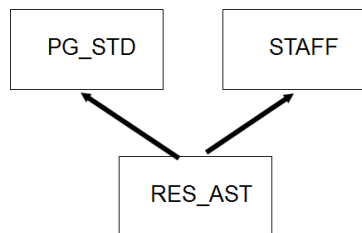
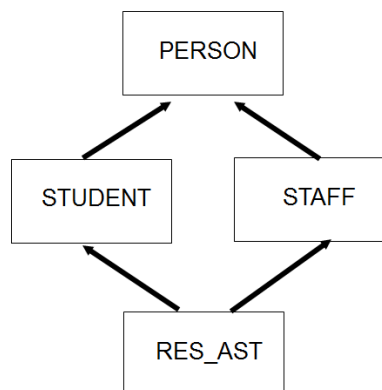
- A container/template/blue-print for objects
- **Objects in a class are called instances**
- Class is also an object with own class attributes and class methods

Class and Objects



Inheritance

- Subclasses, Superclasses, and Inheritance
 - ✓ Inheritance is a special type of relationship between classes; the inheriting class inherits the properties of the base class
 - ✓ Special cases are subclasses and more general cases are superclasses
 - ✓ Process of forming a superclass is generalization; forming a subclass is specialization
 - ✓ Subclass can define its own unique properties as well
 - ✓ Subclass can redefine inherited methods.

Single Inheritance**Multiple Inheritance****Repeated Inheritance**

Overriding, Overloading, and Polymorphism

- Overriding
 - ✓ Process of redefining a property within a subclass.
- Overloading
 - ✓ Allows name of a method to be reused with a class or across classes.
- Polymorphism
 - ✓ Means 'many forms'. Three types: operation, inclusion, and parametric.

Example of Overriding

- We write the fee calculation method for students, that is based on credit hours enrolled

```
method void calcFee (int crHrs) {
    fee = crHrs * 3000; }
```
- We give a discount of 25% to research associate for the support that he/she provides

```
method void calcFee (int crHrs) {
    fee = crHrs * 3000;
    fee = fee - fee * .25}
```

Complex Objects

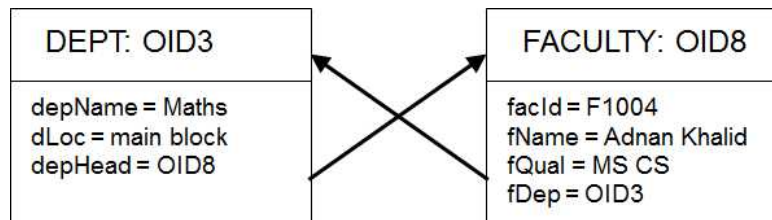
- An object that consists of sub-objects but is viewed as a single object.
- Another special type of relationship between objects, called IS_PART_OF

Modeling Complex Object

- Contained object can be encapsulated within complex object, accessed by complex object's methods.
- Or have its own independent existence, and only an OID is stored in complex object.

Relationships

- Relationships represented using reference attributes, typically implemented using OIDs.
- Consider how to represent following binary relationships according to their cardinality: 1:1 , 1:* , *:*
- 1:1 Relationship Between Objects
 Add reference attribute to A and, to maintain referential integrity, reference attribute to B.



- 1:* and *: Relationships

Add reference attribute to B and attribute containing set of references to A.

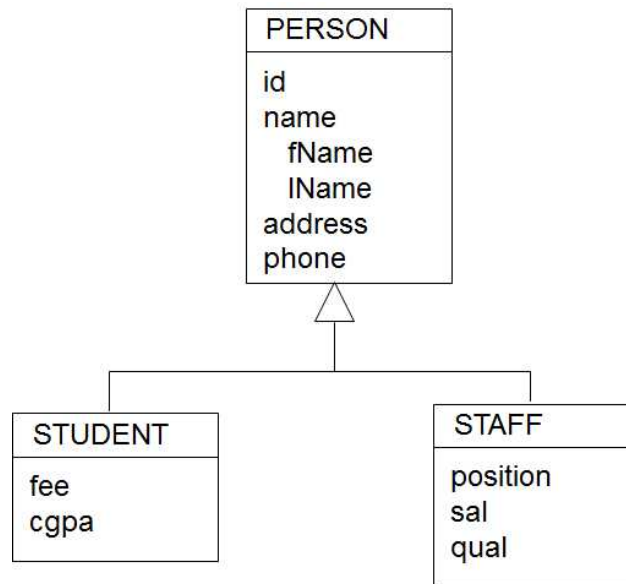
Add set of references attribute to both A & B.

References are maintained by system itself.

Implementing OODBs

- Object-Orientation features have been mentioned.
- One possible way of implementing OODBs is to transform objects into relations.
- Using SQL to perform operations on objects.

Storing Objects in RDBs



Approach

- Each class or subclass converted to a relation
 PERSON (id, fName, lName, address, phone)
 STUDENT (id, cgpa, fee)
 STAFF (id, qual, sal, position)
- Each subclass converted to a relation
 STUDENT (id, fName, lName, address, phone, cgpa, fee)
 STAFF (id, fName, lName, address, phone, qual, sal, position)
- Map the entire hierarchy to a single relation
 PERSON (id, fName, lName, address, phone, cgpa, fee, qual, sal, position)
 SQL operations need to be adjusted accordingly for the insertion/deletion operations on objects.

Lecture -4**Previous Part of Course**

- Factors that necessitated the OODBMSs
 - ✓ Demand of certain applications.
 - ✓ Limitations of the RDM.
 - ✓ Success of Object-Orientation as a Programming Paradigm.

Focus of this Part

- One approach adopted by industry/academia towards OODBMSs.
- Relational Extensions.

Reference

- Chapter 4 of C. Prabhu, “Object-Oriented Database Systems: Approaches and Architectures”, Third Edition, Prentice-Hall of India.

The Basic Theme

- Enjoy the benefits of both worlds.
- Stick to the basic relational model.
- Have the OO features like complex objects, UDTs etc

Design Techniques

- In mid 80s, Stonebraker presented different proposals.
- The Extended Type System for a OODBMS should allow:
 - ✓ Definition of new data types.
 - ✓ Definition of new operations of so defined data types.
 - ✓ Implementation of access methods.
 - ✓ Optimized Query Processing for queries on new data types.

Relational vs OO DBMS

- Huge debate in favor of both.
- Caution: Be careful while comparing.

Extensions Techniques in RDBMSs

- Different steps adopted by different DBMSs, like
 - ✓ Support for variable length “undefined” data values.
 - ✓ Using this support, generalized user-defined data types can be represented.
- Like Oracle supported RAW, LONG and LONGRAW (up to 65535 bytes).
- Sybase supported TEXT and IMAGE up to 2GB, and also others.
- These features were partial support for storing complex data.
- Another type of extension is TEXT and BULK (binary) supported by user-defined procedures that can operate on these types.

- Such facilities were mainly used to capture non-text data, like voice, medical charts, finger prints.
- Then user-defined keywords are associated with user-defined procedures.
- These keywords and the specification of the procedures are stored in tables.
- The user calls the procedure and table is referred to resolve the call.
- For example, consider the function.
integer dist(d1, d2)
- The possible usage of this function may include.
Select dist(x, y) from T.

- **Example**

$x(4, 5), y(7, 9)$

$\text{dist} = \text{Sqrt}((x_1 - x_2)^2 + (y_1 - y_2)^2)$

$\text{dist}(x, y) = \text{Sqrt}((4 - 7)^2 + (5 - 9)^2)$

$\text{dist}(x, y) = \text{Sqrt}(9 + 16) = 5$

Select from T where $\text{dist}(x, y) = \text{expr}$

Lecture -5**Focus of this Part**

- One approach adopted by industry/academia towards OODBMSs
- Relational Extensions
- General Approaches/Tactics
- Two Case Studies

POSTGRES

- Extended RDM DBMS.
- Developed at UC Berkeley in mid 80s by Prof Stonebraker and his group.
- Commercialized as ILLUSTR.
- Extended INGRES (a RDBMS) to support OO features.
- Basic idea in POSTGRES was to introduce minimum changes in the Codd's original relational model to achieve the objective.
- Advantage is the continuity with the previous product (INGRES) and provision of OO features in the new product.

Design and Architecture of PG

- Design objectives of POSTGRES declared by Stonebraker were:
 - ✓ To Provide better support for complex objects.
 - ✓ Support for UDT, operators and access methods.
- Design objectives of POSTGRES declared by Stonebraker were:
 - ✓ Facilities for active databases, and inferencing.
- PG provided extensions to define ADTs, that can be used as types attributes.

Data Definition in POSTGRES

- QUEL provided in INGRES.
- POSTQUEL in POSTGRES.
- Most of QUEL commands are included in POSTQUEL.
- However, many extensions have been included, like:
 - ✓ Complex objects.
 - ✓ UDTs and their access methods.
 - ✓ Time varying data (snapshots and historical data).
 - ✓ Iterative queries.
 - ✓ Alerters, triggers and rules.
- Built-in Data Types include:
 - ✓ Integer, Floating Point, Fixed Length Character String.
 - ✓ Unbounded Varying Length Arrays. with Arbitrary Dimensions.
 - ✓ **POSTQUEL.**
 - ✓ Procedure.

- Conventional dot notation for referencing, EMP.Name
- POSTQUEL type fields can contain sequence of data manipulation command.
- Procedure type can contain procedures written in General Purpose High Level Language.
- Fields of type POSTQUEL and procedure can be executed using EXECUTE command.
- For example: EMP(name, age, salary, hobbies, dept) where hobbies is of POSTQUEL type.
- This field can store queries to retrieve hobbies from other relations, like:
 - EXECUTE (EMP.hobbies)
 - WHERE EMP.Name = "Izzat Gul"
- Comparison!!.

Complex Objects

- Type POSTQUEL can store shared complex objects and multiple representations of data.
- It can contain sequence of commands to fetch data from other relations that represent sub-objects.
- Consider Simple objects
 - ✓ POLYGON (id,)
 - ✓ CIRCLE (id,)
 - ✓ LINE (id,)
- A complex object can be defined as:


```
CREATE OBJECT (name = char[10], obj = postquel)
```
- The values of objects can be

object(a)	RETRIEVE (POLYGON.all) RETRIEVE (CIRCLE.all)	where POLYGON.id = 10 where (CIRCLE.id = 40)
object(b)	RETRIEVE (LINE.all) RETRIEVE (POLYGON.all)	where LINE.id = 25 where (POLYGON.id = 10)
- Multiple representations help in placing appropriate data structure for caching, on the other hand maintaining relational structure for ease of access.
- Transformation are supported by defining a procedure stored in database.
- Create OBJECT (name = char[10], obj = postquel, display = cproc)
- cproc is a procedure written in C language that displays the object.
- But storing procedure for every object is wastage; solution is to store in separate relation and object is passed as parameter.
- To execute the procedure
 - execute (obj.proc) with ("apple") where obj.value = 'display-list'
- Comparison!!.

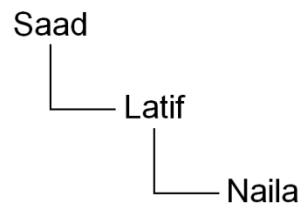
Time Varying Data and Versions

- Historical data access and versions supported.
- Normal queries access current data.
- Historical access: retrieve (E.all) from E in EMP
[“March 28, 2009”]
- E is called the tuple variable
- Square brackets represent timestamp.
- System will search back through historical data to retrieve the relevant tuples.
- The RETRIEVE INTO command will materialize the snapshot.
- Same effect through DISCARD command; it provides a cutoff point.
- Like discard EMP before “one year” deletes all data older than one year.
- It also supports specification of range of timestamps:
relation name [date1, date2]
- RETRIEVE (E.all) from E in EMP
[“Mar 10, 2009”, “Nov 1, 2009”]
- Either of these limits can be omitted.
- RETRIEVE (E.all) from E in EMP
[, “Nov 1, 2009”]
- RETRIEVE (E.all) from E in EMP
[“Mar 10, 2009”,]
- Memory hierarchy of POSTQUEL is:
 - a) main memory
 - b) secondary memory (HD)
 - c) tertiary memory (optical disk)
- Current data in secondary memory and tertiary memory used for historical data.
- POSTGRES also supports versions.
- Version can be created from a relation or a snapshot.
- Updates in version will not be reflected in relation.
- Updates in relation will be conditionally visible in versions.
- If user wants constant version then the version should be created from snapshot.
- Version creation example:
 - NEW VERSION EMPVER from EMP
- Version changes into Relation:
 - MERGE EMPVER into EMP

Iterative Queries

- Iterative query is a special query that contains sub-queries inside or a query that is repeated multiple times.
- Required to support transitive closure.

- Suppose a relation in Employee Hierarchy:



- EMP(name char[20], mgr char[20])
RETRIEVE * into SUBORDS (E.name, E.mgr) from E in EMP, S in SUBORDS where
E.Name = "Saad" or
E.mgr = S.name
- Command executes till there are no changes in SUBORDS.

Lecture -6**Iterative Queries**

- Required to support transitive closure.
- To get the subordinates of an employee.
- Command used is 'retrieve *', * makes it iterative.
- For a relation
EMP(name char[20], mgr char[20])
RETRIEVE * into SUBORDS
(E.name, E.mgr) from E in EMP, S in
SUBORDS where
E.Name = "Saad" or
E.mgr = S.name
- Command executes till there are no changes in SUBORDS.
- Other commands where * can be applied:

✓ APPEND

✓ DELETE

✓ EXECUTE

✓ REPLACE

✓ RETRIEVE INTO

Alerters and Triggers

- Can be specified using 'always'.
RETRIEVE always (EMP.all)
where EMP.name = "Saad"
- Trigger is an update query.
DELETE always DEPT dname where
count (tmp.name by DEPT.name) = 0
- Forward chaining is supported by ALWAYS; an update wakes up a chain of alerters, triggers and commands.
- Defining ANCESTOR on base relation.
PARENT(parent-of, offspring)
Range of A is ANCESTOR.
Range of P is PARENT.
DEFINE view ANCESTOR (P.all)
DEFINE view * ANCESTOR (A.parent-of, P.offspring) where A.offspring = P.parent-of
- Query on this view may involve recursive execution of commands.

Compilation and Fast-Path

- Provide performance enhancement in POSTQUEL.
- Any command can be compiled.
- Stored in a system catalogue.
CODE (id, owner, command)
- Fast-path further improves it.

Conclusion

- POSTGRES, first extended relational DBMS.
- Provided OO features like:
 - ✓ Complex objects, triggers, alerters, versions, historical data and inference rules etc.
- Enhancement in basic relational model.

STARBURST DBMS

- SB project initiated in 1985 at IBM research lab.
- Problems to address:
 - ✓ Manage the complexity of typical and advanced DB applications.
 - ✓ Existing products seemed saturated for extensions.
 - ✓ Users influenced by OO features demanded them in DBMSs.
 - ✓ Cost effectiveness of distributed processing became common.
 - ✓ Expert system features required by users.
- Objective: build from scratch as an extensible DBMS prototype having the capabilities:
 - ✓ A product that facilitates extensions.
 - ✓ Provide a test-bed for DBMS experiments.
- Basic philosophy of SB: the varying/conflicting requirements can be met by extending a basic data model.
- It is also basically a relational data model DBMS providing extensions to relational language.
- Built originally in C, later C++.
- Works on different platforms.

Design and Architecture of SB

- A table can be stored in variety of methods (such as heap files, non-recoverable temporary file, B+ trees etc).
- Each table has associated attachment types.
- Integrity constraints are checked by providing veto power to attachment.
- Storage and attachment methods utilize common services:
 - ✓ Concurrency Control
 - ✓ Event Queues
 - ✓ Recovery
- The core provides a standard event queue for different activities.
- Several extensions are made to SQL in SB.

UDTs and Functions

- Fixed DTs and functions was a major limitation of traditional DBMSs.
- OODBMSs have endeavored.
- STARBUSRT has functions:
 - ✓ Scalar functions.
 - ✓ Set predicate functions.
 - ✓ Aggregate functions.
 - ✓ Table Functions.

STARBURST Type System

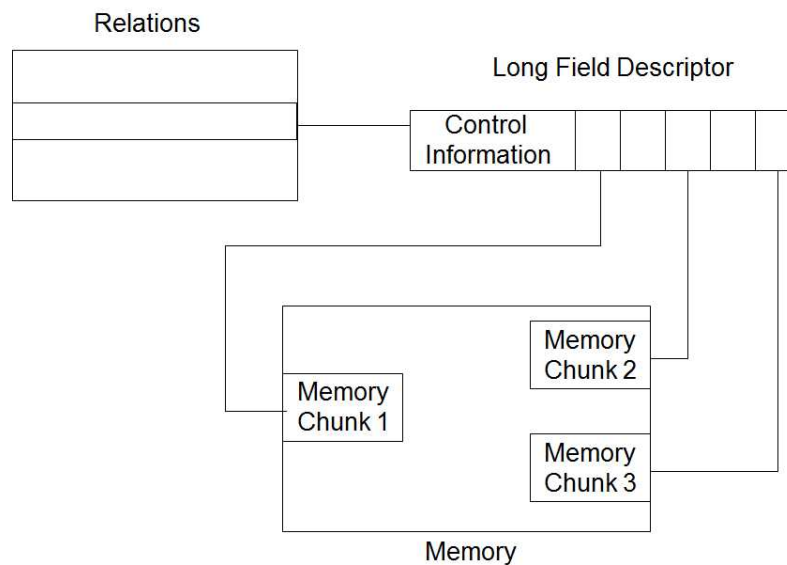
- SB supports hierarchical type system in which UDTs can be encapsulated.
- Inheritance supported in this hierarchy.
- Type checking at compile time.
- Impedance Mismatch.
- Type system of DBMS is different from PL:
 - ✓ They are persistent.
 - ✓ DBMSs share facilities.

SB Type System & Functions

- SB evaluates the command to link it to appropriate functions.
- For embedded commands, pre-processor identifies DB commands, performs the type transformations and calls the code.
- So SB types retain their representation and behavior.

Lecture -7**Complex Objects in SB**

- Not supported by conventional relational DBMSs.
- Normally represented as
 - ✓ Long field.
 - ✓ By composing other atomic object rows; called XNF in SB.
- Long Field Manager stores objects up to 1.5 GB.
- LFM controls transactions on long field.
- Stores long fields in small holes and using algos and control structure called Long Field Descriptor.
- SB creates an attachment for the table that has a long field.

Long Field Manager (LFM)**Structure of Objects in SB**

- A complex object comprises multiple sub-complex or atomic objects.
- Typically, of a hierarchical nature; repetition and sharing of components.
- A transaction has to process the entire hierarchy.
- SB gives the concept of XNF/
- **A complex or composite object is defined as a single object composed of:**
 - ✓ Many sub-objects of different types.
 - ✓ Relationships with other objects.
 - ✓ Each object may have multiple instances.
- Examples: country, engine, PC ...

Implementation of Complex Objects in SB

- Two approaches in SB for the implementation of complex objects:
 - ✓ Navigational
 - ✓ Nested Relations
- Nested Relations
 - ✓ The nested relations (NF2 relations) approach places the objects in hierarchy together.
- SB supports complex objects that can be shared within and among objects.
- Containment of objects may be infinitely deep.

SB XNF

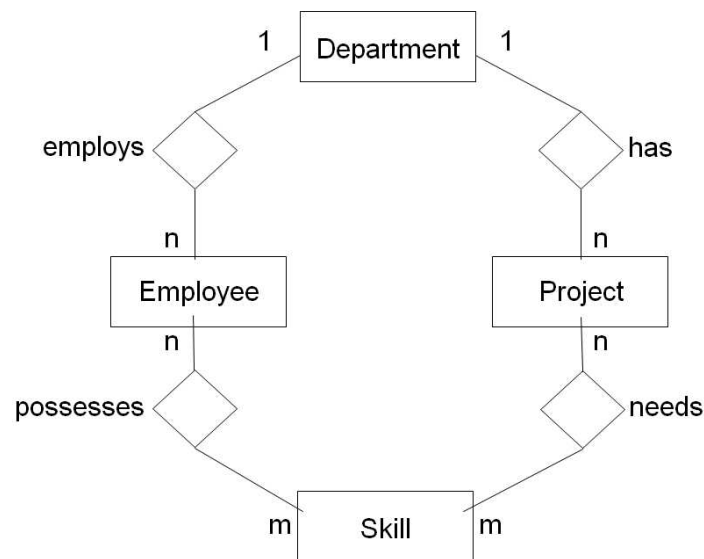
- XNF is based upon the concept of relational view; a virtual table whose definition is stored in DB
- The table that it produces is not stored.
- Definition is basically a query over stored tables or views (intentional).
- XNF view consists multiple tables of an organized structured heterogeneous collection of interrelated rows.
- In contrast with navigational and NF2 approaches.
- The XNF approach is different from Navigational and NF2 approaches !
- XNF objects can be used in queries; can be combined, projected and restricted
- Combined: by defining relationship of any node of one with root of other.
- Restriction: by applying additional predicates on nodes.
- SB has a separate query processor; transforms XNF query to standard internal representation.

Lecture - 8**Previous Lecture**

- SB discussion
- LFM
- XNF
- Complex object representation approaches
- XNF relational views

Representing Complex Object

- XNF query identified by OUT OF and consists of:
 - ✓ Definition of component tables (identified by SELECT)
 - ✓ Relationships between component nodes by RELATE <nodes> WHERE <predicate>
- The resulting nodes and relationships form a directed graph connected to a root.

**Definition of Complex Object**

CREATE VIEW A-Dept-obj AS

OUT OF

A-depts AS (SELECT * FROM department WHERE DNO = 'D101')
 A-emps AS (SELECT * FROM employee)
 A-projs AS (SELECT * FROM project)
 A-skills AS (SELECT * FROM skills)
 employs AS(RELATE A-depts d, A-emps e WHERE d.dno = e.deptno),
 has AS(RELATE A-depts d, A-projs p WHERE d.dno = p.deptno)
 possesses AS(RELATE A-emps e, A-skills s WHERE e.skillno = s.sno)
 needs AS(RELATE A-projs p, A-skills s WHERE p.skillno = s.sno)

Active DB Features in SB

- Active DB allows users to define rules that are invoked whenever specific condition occurs, irrespective of application.
- For enforcing integrity constraints, maintaining derived data, building interfaces to KBS.
- Active DB Features in SB & Other DBMS
 - ✓ Other similar databases respond to row level changes only.
 - ✓ The rules in SB respond to aggregate or accumulative changes in database.
- SB adopts two approaches for implementation:
 1. Relational oriented production rule system that monitors set of changes to base tables.
 2. Called Alert monitors objects and invocation of applications.
- Production rule based system: comprises a stored table storing
 - ✓ Trigger clause (SQL operation(s))
 - ✓ Conditional clause (IF-SQL query)
 - ✓ Action clause (any seq of DB commands)
- Action part can suppress the changes of transaction or can perform further modifications.
- These modifications can trigger other rules; so forward chaining is initiated.
- Rules can be activated or de-activated any time.
- Example of Defining Rule


```
CREATE rule non-emp-dep ON department
  WHEN DELETED
  IF 'SELECT * FROM Employees
      WHERE Deptno IN
      (SELECT Dno FROM dd AS (DELETED()))'
  THEN 'SELECT d.Dno, 'non-empty'
      FROM d AS (DELETED())
      WHERE d.Dno IN (SELECT Deptno
      FROM Employees)' 'ROLL BACK WORK'
```
- Rules can be partially ordered using PRECEDES or FOLLOWS clauses.
- Rules stored in system catalogue.
- CC mechanism is used when rules are created, mod or del.
- Second approach: Alert trigger system.
- Allows users to define rules at a higher level of abstraction.
- Active DB Features in SB Example
- Like tour expenditures of an employee may need to adjust budget and some other data.
- This is implemented by defining a method on application specific object or XNF view; a virtual table rather than actual table.
- To track activation of methods, an active table is created that is append-only.
- Alert rule is created by defining an SQL row; a named SQL query.
- FROM and WHERE clauses express rule's condition.

- SELECT contains rule's actions including invocation of some functions.
- For the tour expenditure example, method may add name, expenses and other details of employee raising the bill.
- Alerts need to be explicitly activated unlike production rules.

Conclusion

- SB is based on Extended Relational DM, provides complex objects by relational views.
- Adv: upward compatibility of existing relational applications.

Object/Relational Systems

- Similar Objectives as in Extended Relational.
 - ✓ Providing OO capabilities to RDBMS technology.
 - ✓ Preserving full relational capabilities.
- Approaches taken:
 - ✓ Building object layer on top of conventional relational system, like Open ODB/ODAPTER.
 - ✓ Building a new DBMS from foundation, like ILLUSTRATE or UNI SQL/X.
- Built on different architectures, like query server architecture or client/server architecture.

Open ODB/ODAPTER

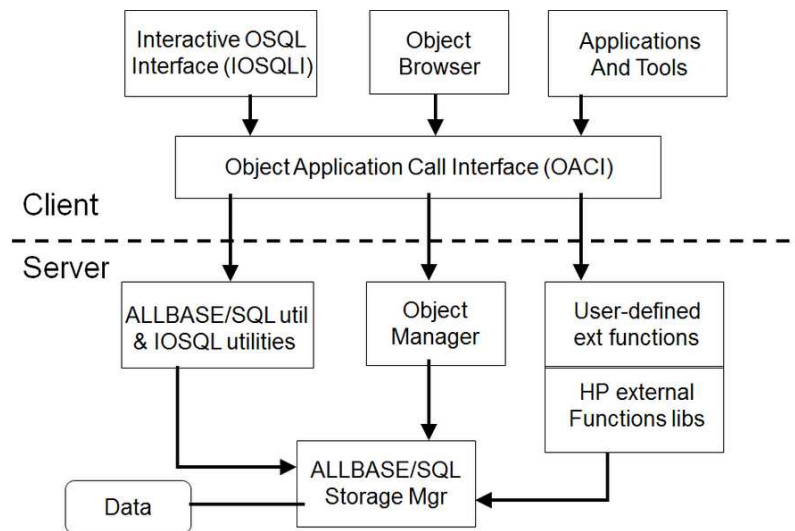
- Open ODB an ORDBMS from HP during mid-90s aims to support broad base of applications.
- Based on Iris; C/S architecture.
- Both data & applications can be shared by users and apps.
- Clients use API to access info.
- Data access is through OSQL.
- Open ODB uses relational techniques to support OO.
- Object model implemented by an object manager.
- Mapping of OO schema and queries to Relational ones.
- The underlying relational storage manager/DBMS for Open ODB is ALLBASE/SQL
- ODAPTER, a later version, relaxes this binding, allows any DBMS.

Object Data Model and Language

Open ODB – Data Model

- Data model comprises:
 - ✓ Object
 - ✓ Type
 - ✓ Function
- Object
 - ✓ Every object comprises data and stored code that can operate on data.
 - ✓ Every object has OID
- Type
 - ✓ All user defined types are sub-types of type 'user surrogate'; Every object has OID.

- ✓ Objects with same properties are grouped into types.
- Functions
 - ✓ Properties of objects, relationships between objects and operations between them are implemented through functions.
 - ✓ The function accepts an object as input and produces one (that can be aggregate object)
 - ✓ Function overloading is allowed.
 - ✓ Inheritance through is-a relationship between objects, like:
 - CREATE TYPE student FUNCTIONS (Roll No Char(20))
 - CREATE TYPE CS-Student SUBTYPE of student.
- Functions implemented as
 - ✓ Stored functions (OSQL).
 - ✓ External functions (any GP PL).
- External functions are also used to access data and code from outside open ODB.
- It helps in integration with other applications.

Lecture -9**Open ODB Architecture****Open ODB Architecture Components on Client Side**

- IOSQLI: allows interactive use of OSQL.
- OB: a graphical browsing interface to open ODB database; allows dynamic updates to data and schema.
- OACI: allows developing and linking applications in any PL, like C++, Pascal, FORTRAN etc; the OSQL statements passed as arguments.
- Application and Tools.

Open ODB Architecture Components on Server Side

- OM: main processor; handles OSQL calls; passes Open ODB clients' calls to SM or to external function/library.
- SM: of ALLBASE/SQL; performs all DB activities.
- EFL: applications encapsulated through EFs; of two types; user defined and system defined; can be called in OSQL statements.

Transaction Management and Interfaces in ODB

- TM in Open ODB performed conventionally; 3 isolation levels:
 - ✓ Repeatable read.
 - ✓ Read committed.
 - ✓ Read uncommitted.
- Log file for Logging recovery.
- Indexing on OID or user defined; automatically or on demand.
- Clustering of related functions for improved performance.
- Authentication and access control for group and individual users.

- Storing unformatted large objects.
- Dynamic schema modification allowed.
- An object can belong to multiple types; so dynamic type change.

ODB Interface With C++

- C++ interface to ODB provides:
 - ✓ Session control facilities.
 - ✓ Execution OSQL commands.
 - ✓ To define cursor.
 - ✓ To associate C++ object with OSQL function.
 - ✓ Use C++ classes that represent OSQL data types.
- O-ODB users can define User-defined types in C++.

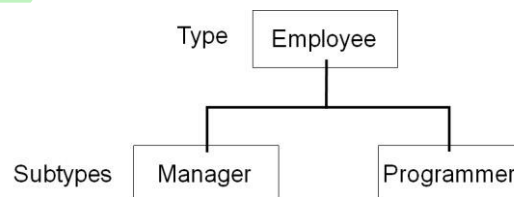
ODB Interface With Other Languages

- Smalltalk interface also provided in O-ODB and also for many others.

OSQL Features

- Excels SQL to define object types and classes.
- User defined and system functions can appear in Select and Where clauses.
- OSQL is a complete language:
 - To manipulate database.
 - Grant authorizations.
 - Define transactions and admin etc.

Example OSQL Implementation



- Example of function that implements the rule of converting a manager to a programmer who have less than a certain number of employees:
- CREATE TYPE employee FUNCTIONS (name, CHAR, salary, FLOAT, picture, binary)
- CREATE TYPE manager SUBTYPE of employee FUNCTIONS (manages SETTYPE (Employee))
- CREATE OBJECT as Programmer FUNCTIONS (Name, salary, Languages)
 - :shah ('MA Shah', 80000, SET ('C++', 'VB')), :papu ('RS Papu', 75000, SET ('C#'));

Example OSQL Implementation Changing Object Type

- CREATE FUNCTION Mgr-to-Pgr (INTEGER minemps) → BOOLEAN AS OSQL
 - BEGIN
 - For m IN Manager Do

```
If ( COUNT (Manager(m)) < minemps ) THEN
```

```
BEGIN
```

```
ADD TYPE Programmer to m;
```

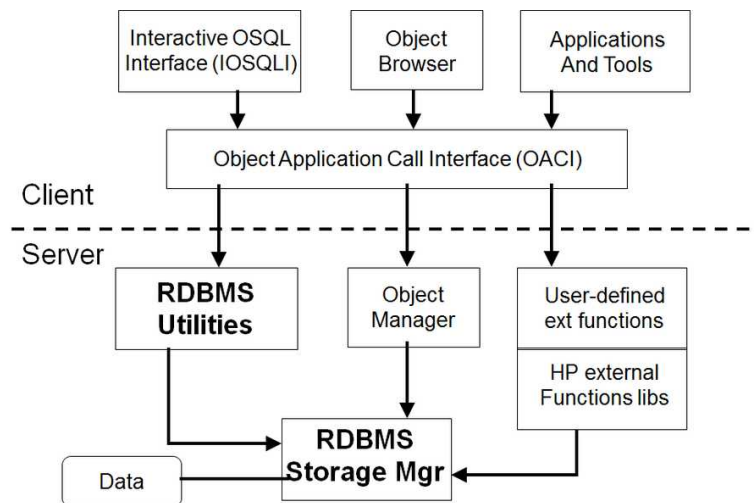
```
REMOVE type Manager FROM m;
```

```
ENDIF
```

```
END
```

- Example call to function: CALL Mgr-to-Pgr (3)

ODAPTER Architecture



- New version of open DB.
- Claimed to be used with 10 different RDBMSs.
- Interfaces to Smalltalk and C++ provided.

ILLUSTRA DBMS

- The final name of PG when launched as a commercial ORDBMS in mid 90s.
- Supports all features of PG in a better way, like:
 - ✓ Type inheritance.
 - ✓ UDTs.
 - ✓ Ad-hoc queries etc.
- Supports SQL-3 standard.
- Functions written in C or SQL.
- ILLUSTRA can be extended with new access methods; query processor decides which access method to be invoked.
- Available on different major platforms.

Data Types in ILLUSTRA

- Create type circle (

```

        internal length = 30
        input = circle-in
        output = circle-out
    ); // a base type.

```

- The attributes of the tables can be of any type; UDT or built-in.
- UDTs may be of base type or composite type.

Functions in ILLUSTR

- Input function converts an external representation to internal one and other way round.
- Create function circle-in (text)


```

                return circle-out
                as external name '/temp/circle-so'
                language C;
            
```
- Create function circle-out (circle-t)


```

                return text
                as external name '/temp/circle-so'
                language C;
            
```

DBMS Covered

- O-ODB
- ODAPTER
- ILLUSTR

Lecture -10**ILLUSTRA DBMS - Commands**

- Table creation
CREATE table employees (name text, company text)
- Columns can be base or composite type.
- System columns are also added, like OID, commit times, relative tuple id etc.
- Inheritance also applies to table, like:
CREATE table employees of type
employee-t under people
- Query on supertable also fetches the data from subtable(s).

ILLUSTRA Architecture

- Comprises three main comp:
 - ✓ Main Server DB Engine
 - ✓ Visualization Tool
 - ✓ Datablades
- Server – enhances the typical table with OO features; queries on objects transformed into those on relations
- VT – for better visualization of objects.
- Datablades – provide manipulation for specific applications; combined with a razor (a DBMS) perform specific functions; examples.
- Image datablade can display, transform, resize etc different images.
- Spatial datablade provides points, lines, polygons and their relationships; similarly text datablade provides full text search in an SQL query; like.
- SELECT From Employees WHERE Contains (resume, 'SQL' or 'Quel Programming')
AND (dept = 'computer division')
- LibMI is API interfaces application and DB server.
- Apps can be seen/executed from client or server.
- Support for large data objects (BLOBS):
 - ✓ External file; used for images etc.
 - ✓ Large-object; handled through handles.
 - ✓ Large-text.

Other ORDBMS Systems

- | | |
|------------------------------------|---------------|
| • ORACLE 8i (Start of OO features) | • IBM's DB2 |
| • PERSISTENCE | • DAFFODIL DB |

ORDBMS - Conclusion

- Object-Relational and Extended Relational approaches have a common underlying philosophy with a minor difference.
- Here we discussed different products like O-ODB, ODAPPER, ILLUSTR.

Part-4

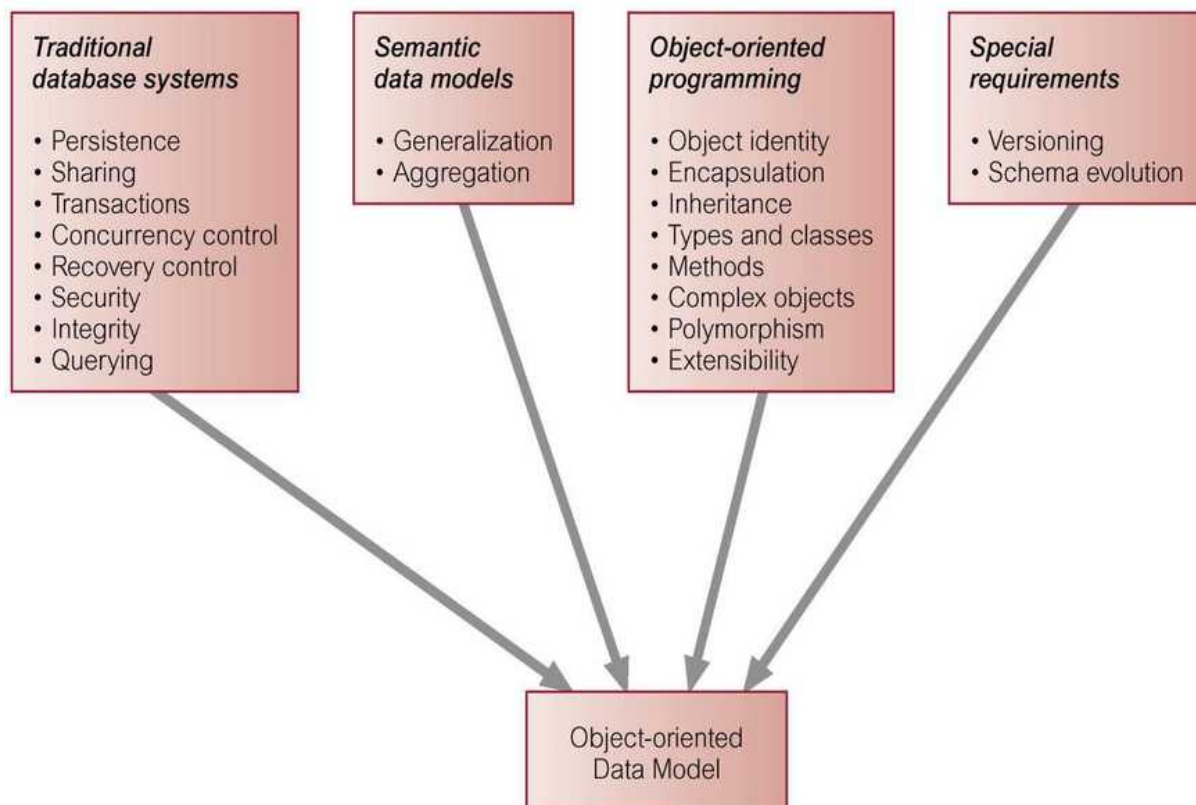
- OODBMS Concepts.
- Ref 1: Ch-26, T. Connolly, C. Begg, “Database System: A Practical Approach to Design, Implementation and Management”, 4th Edition, Pearson Education.
- Ref 2: Ch-3, C. Prabhu, “Object-Oriented Database Systems: Approaches and Architectures”, Third Edition, Prentice-Hall of India.

Object-Oriented Data Model

- There was no agreed upon standard to define OODBMS.
- Some definitions from KIM (1991):
- Object-Oriented Data Model (OODM)
 - ✓ Data model that captures semantics of objects supported in object-oriented programming.
- There was no agreed upon standard to define OODBMS.
- Some definitions from KIM (1991):
- Object-Oriented Database (OODB)
 - ✓ Persistent and sharable collection of objects defined by an ODM.
- Object-Oriented DBMS (OODBMS)
 - ✓ Manager of an ODB.
- ZDONIK and MAIER present a threshold model that an OODBMS must, at a minimum, satisfy:
 - ✓ It must provide database functionality.
 - ✓ It must support object identity.
 - ✓ It must provide encapsulation.
 - ✓ It must support objects with complex state.
- Inheritance may be useful not must
- KHOSHAFIAN and ABNOUS define OODBMS as:
 - ✓ OO = ADTs + Inheritance + Object Identity
 - ✓ OODBMS = OO + Database Capabilities
- PARSAYE *et al.* gives:
 1. High-level query language with query optimization.
 2. Support for persistence, atomic transactions: concurrency and recovery control.
 3. Support for complex object storage, indexes, and access methods.
 4. OODBMS = OO System + (1), (2), and (3).

Lecture -11**Some Commercial OODBMSs**

- GemStone from Gemstone Systems Inc.
- Objectivity/DB from Objectivity Inc.
- O2 from O2 Technology Corporation.
- ObjectStore from Progress Software Corp.
- ObjectStore from Progress Software Corp.
- Ontos from Ontos Inc.
- FastObjects from Poet Software Corp.
- Jasmine from Computer Associates/Fujitsu.
- Versant from Versant Corp.

Origins of the Object-Oriented Data Model**Functional Data Model**

- Interesting because it shares certain ideas with object oriented approach including OID, inheritance, overloading, and navigational access.
- In FDM, any data retrieval task can viewed as process of evaluating and returning result of a function with zero, one, or more arguments.
- In the FDM, the main modeling primitives are entities and functional relationships.

- Entity: Decomposed into abstract entity types and printable entity types.
- Entity types correspond to classes of 'real world' objects and declared as functions with zero arguments that return type ENTITY.
- For example:
 - Staff() → ENTITY
 - PropertyForRent() → ENTITY

Printable Entity Types

- Printable entity types are analogous to base types in a programming language.
- Include: INTEGER, CHARACTER, STRING, REAL, and DATE.

Attributes

- An attribute is a *functional relationship*, taking the entity type as an argument and returning a printable entity type.
- For example:
 - staffNo(Staff) → STRING
 - sex(Staff) → CHAR
 - salary(Staff) → REAL

Composite Attributes

- Name() → ENTITY
- Name(Staff) → NAME
- fName(Name) → STRING
- lName(Name) → STRING

Relationships

- Functions with arguments also model relationships between entity types.
- Thus, FDM makes no distinction between attributes and relationships.
- Each relationship may have an inverse relationship defined.
- For example:
 - Manages(Staff) —» PropertyForRent
 - ManagedBy(PropertyForRent) → Staff
 - INVERSE OF Manages
- Can also model *:.* relationships:
 - Views(Client) —» PropertyForRent
 - ViewedBy(PropertyForRent) —» Client
 - INVERSE OF Views
- For example:
- And attributes on relationships:
- viewDate(Client, PropertyForRent) → DATE

Inheritance and Path Expressions

- Inheritance supported through entity types.
- Principle of substitutability also supported.

Staff() → ENTITY

Supervisor() → ENTITY

IS-A-STAFF(Supervisor) → Staff

Other Features

- Derived functions can be defined from composition of multiple functions (note overloading):

fName(Staff) → fName(Name(Staff))

fName(Supervisor) → fName(NAME(IS-A-STAFF(Supervisor)))

Declaration of FDM Schema**Entity type declarations**

Staff() → ENTITY

Supervisor() → ENTITY

PropertyForRent() → ENTITY

Client() → ENTITY

Name → ENTITY

Attribute declarations

fName(Name) → STRING

lName(Name) → STRING

fName(Staff) → fName(Name(Staff))

lName(Staff) → lName(Name(Staff))

fName(Client) → fName(Name(Client))

lName(Client) → lName(Name(Client))

staffNo(Staff) → STRING

pos tion(Staff) → STRING

sex(Staff) → CHAR

salary(Staff) → REAL

clientNo(C ient) → STRING

telNo(Client) → STRING

prefType(Client) → STRING

maxRent(Client) → REAL

propertyNo(PropertyForRent) → STRING

street(PropertyForRent) → STRING

city(PropertyForRent) → STRING

type(PropertyForRent) → STRING

rooms(PropertyForRent) → INTEGER

rent(PropertyForRent) → REAL

Relationship type declarations

Manages(Staff) → PropertyForRent

ManagedBy(PropertyForRent) → Staff INVERSE OF Manages

Views(Client) → PropertyForRent

ViewedBy(PropertyForRent) → Client INVERSE OF Views

viewDate(Client, PropertyForRent) → DATE

comments(C ient, PropertyForRent) → STRING

Inheritance declarations

IS-A-STAFF(Supervisor) → Staff

staffNo(Supervisor) → staffNo(IS-A-STAFF(Supervisor))

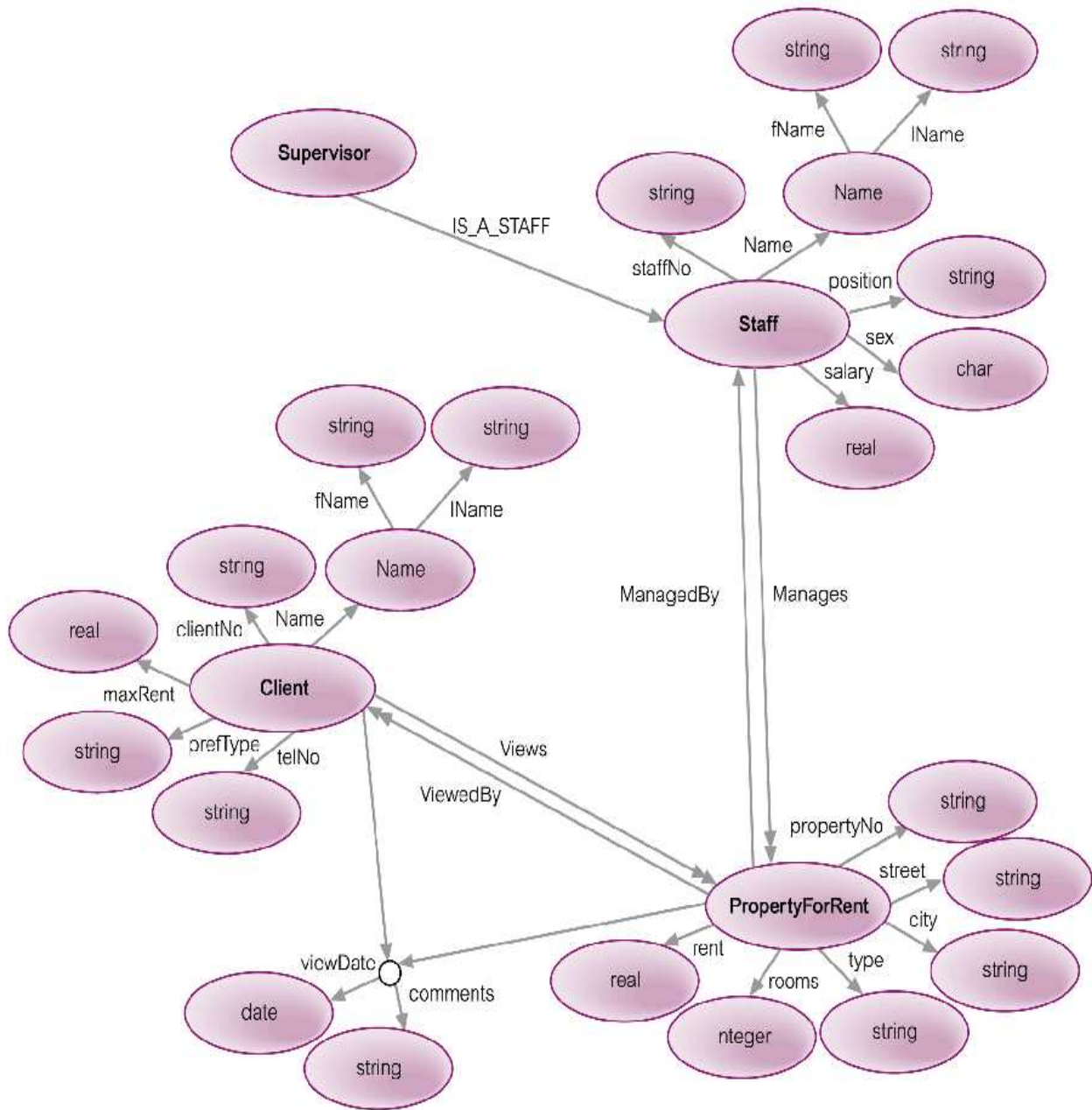
fName(Supervisor) → fName(IS-A-STAFF(Supervisor))

lName(Supervisor) → lName(IS-A-STAFF(Supervisor))

position(Superv sor) → pos tion(IS-A-STAFF(Supervisor))

sex(Supervisor) → sex(IS-A-STAFF(Supervisor))

salary(Supervisor) → salary(S-A-STAFF(Supervisor))

Diagrammatic Representation of Schema**Functional Query Languages**

- Path expressions also used within a functional query.
- For example:

```
RETRIEVE lName(Name(ViewedBy(Manages(Staff)))
WHERE staffNo(Staff) = 'SG14'
```

Or in dot notation:

```
RETRIEVE Staff.Manages.ViewedBy.Name.lName
WHERE Staff.staffNo = 'SG14'
```

Advantages

- Support for some object-oriented concepts.
- Support for referential integrity.
- Irreducibility.
- Easy extensibility.
- Suitability for schema integration.

OODBMS Manifesto – Reference

- Atkinson M., Bancilhon F., Dewitt D., Maier D., Zdonik S. (1989), Object Oriented Database System Manifesto. In *Proc. 1st Intl Conf. Deductive and Object-Oriented Databases*, Kyoto, Japan. 40-57

OODBMS Manifesto

- Support for Complex objects.
 - ✓ Complex object consists of sub-objects.

Lecture -12**OODBMS Manifesto****1. Support for Complex Objects**

- ✓ Obtained by applying constructors on basic objects.
- ✓ Minimal; SET, TUPLE, LIST and ARRAY.
- ✓ Must be orthogonal, so LIST(SET()) is possible.

2. Object Identity

- ✓ Object independent of state.
- ✓ Two objects can be identical or same.

3. Encapsulation

- ✓ Enforced through access through object interface only.
- ✓ One violation required in case of ad-hoc queries; a debate.

4. Types or Classes

- ✓ Either one of them must be supported.

5. Inheritance and Hierarchies

- ✓ Gives concept of super/subclass.
- ✓ Methods rather than rewriting (as in RDM) can be re-used in subclass.
- ✓ Types: Substitution, inclusion, constrain, specialization.

6. Dynamic Binding

- ✓ Overriding, overloading and late binding.
- ✓ Call resolved at run time.

7. Computationally Complete DML

- ✓ To provide support for data processing; SQL-2 was not SQL-99 and onward are.

8. Extensible set of data types

- ✓ Support for UDTs.
- ✓ No distinction with built-in ones.

9. Data Persistence

- ✓ Basic feature required with any DBMS.

10. Managing very large databases

11. Concurrent Users

12. Transaction Management

13. Query Language

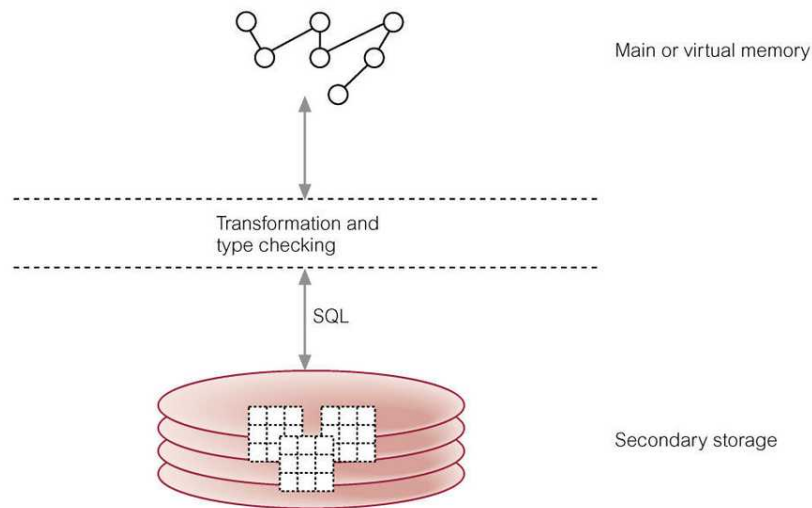
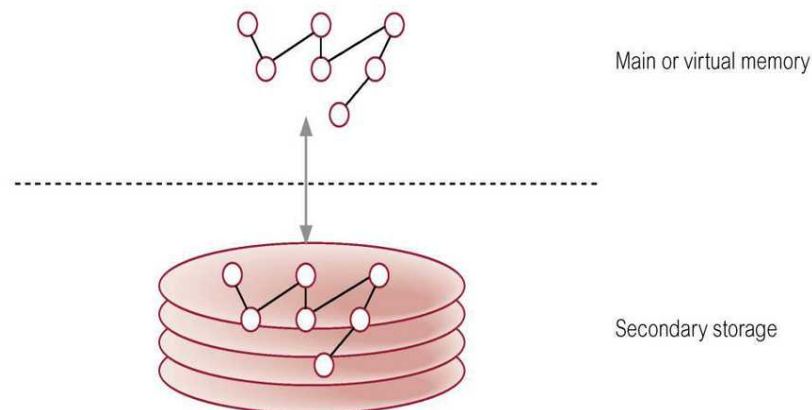
• Optional features:

- ✓ Multiple inheritance
- ✓ Type checking and type inferencing
- ✓ Distributed Database
- ✓ Design transactions (long or nested)
- ✓ Versions

- No direct mention of support for security, integrity, views or even a declarative query language.

OODBMS Perspectives

- Previous OODBMS approaches present a two level storage model.
- Presentation of objects in RAM and Disk is different.
- The conversion needs resources.
- OODBMS gives illusion of single storage model.

Two-Level Storage Model**Single-Level Storage Model**

- Single memory model seems efficient.
- Two types of OIDs involved
 - ✓ Logical ID; independent of physical storage of object.
 - ✓ Physical OID; encodes location.
- OODBMS has to switch between the two.

Pointer Swizzling Techniques

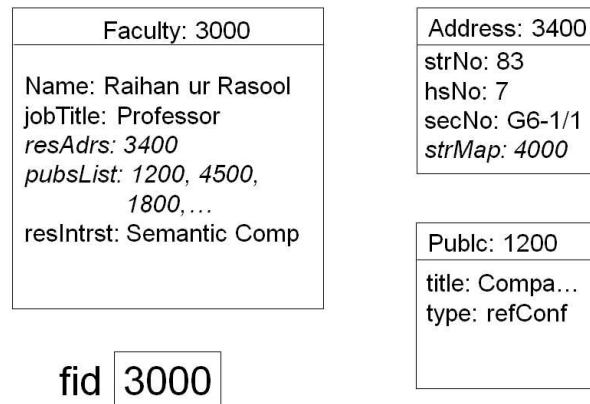
- The action of converting object identifiers (OIDs) to main memory pointers; also called object faulting.
- Aim is to optimize access to objects.

Lecture -13

Previous Lecture

- OODBMS Manifesto
- OODBMS Perspective (Issues)
 - ✓ Memory Management (single/Two levels)
 - ✓ Pointer Swizzling

Pointer Swizzling

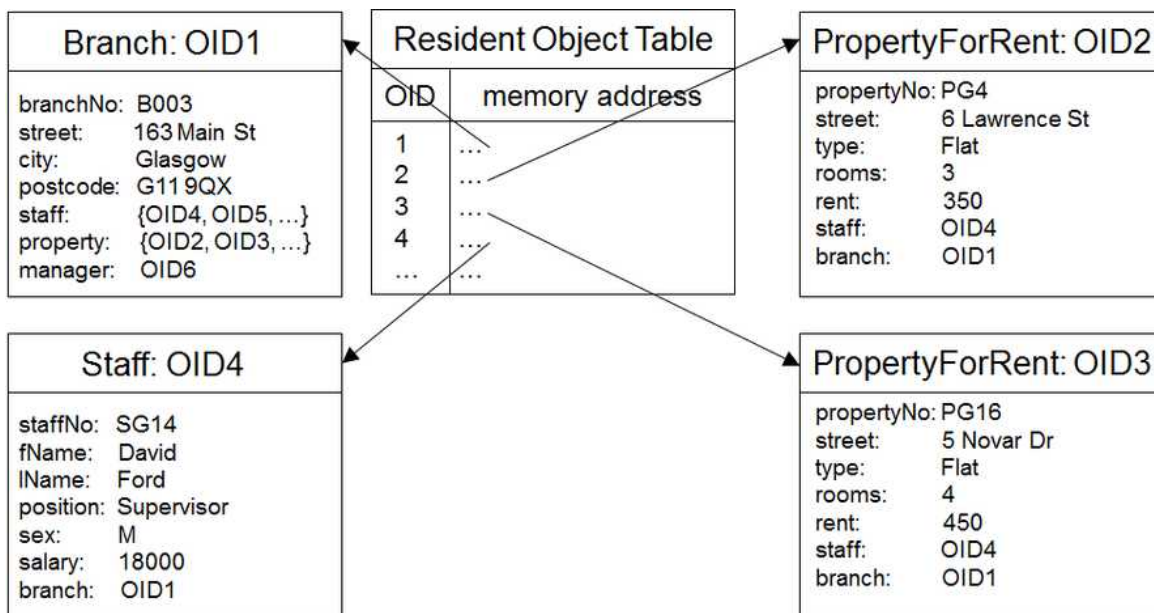


- Once objects have been read into cache, want to record that objects are now in memory to prevent them from being retrieved again.
- Pointer swizzling attempts to provide a more efficient strategy by storing memory pointers in the place of referenced OIDs, and *vice versa* when the object is written back to disk.

No Swizzling

- Easiest implementation is not to do any swizzling.

Resident Object Table (ROT)



Object Referencing

- Need to distinguish between resident and non-resident objects.
- Most techniques variations of edge marking or node marking.

Object Referencing Edge Marking

- Edge marking marks every object pointer with a flag bit:
 - ✓ if bit set, reference is to memory pointer;
 - ✓ else, still pointing to OID and needs to be swizzled when object it refers to is faulted into.

Object Referencing Node Marking

- Node marking requires that all object references are immediately converted to virtual memory pointers when object is faulted into memory.
- First approach is software-based technique but second can be implemented using software or hardware-based techniques.

Other Issues

- Three other issues that affect swizzling techniques:
 - ✓ Eager versus Lazy Swizzling
 - ✓ Copy versus In-Place Swizzling
 - ✓ Direct versus Indirect Swizzling

Eager versus Lazy Swizzling

- An object may contain references of many (simple or complex) objects.

Faculty: OID20
Name: Raihan ur Rasool jobTitle: Professor resAdrs: OID301 pubsList: OID43, OID89, OID900, resIntrst: Semantic Comp

Lazy Swizzling

- Lazy swizzling only swizzles pointers as they are accessed or discovered.
 - ✓ Upon Discovery (copied from field to local variable)
 - ✓ Upon Dereference (actually dereferenced)

Copy versus In-Place Swizzling

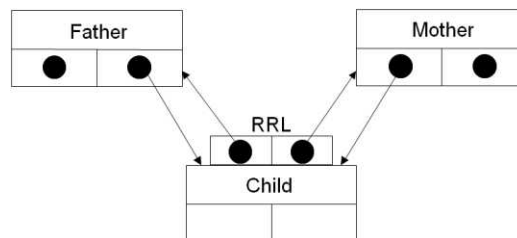
- When faulting objects in, data can either be copied into application's local object cache or accessed in-place within object manager's database cache .
- Copy swizzling may be more efficient as, in the worst case, only modified objects have to be swizzled back to their OIDs.
- In-place may have to unswizzle entire page of objects if one object on page is modified.

Lecture -14**Previous Lecture**

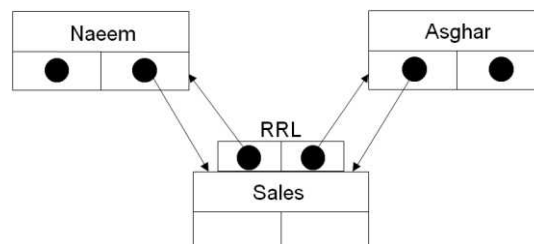
- Pointer Swizzling
 - ✓ No swizzling – needs ROT
 - ✓ Referencing
 - ✓ SW/HW based schemes
- ✓ Eager vs Lazy
- ✓ Copy vs In-place

Direct versus Indirect Swizzling

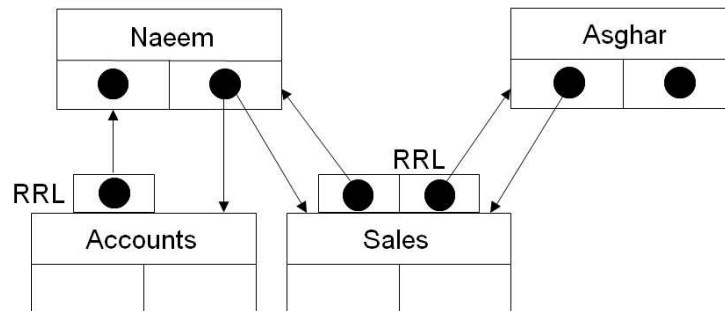
- Only an issue when swizzled pointer can refer to object that is no longer in virtual memory.
- With direct swizzling, virtual memory pointer of referenced object is placed directly in swizzled pointer.
- If an object is displaced from memory, no more in RAM. All references to this object have to be unswizzle.
- Displaced Object's references need to be registered in Reverse Reference List (RRL).

Reverse Reference List (RRL)

- Eager direct swizzling may cause snowball effect on placing and displacing object.
- Maintaining RRLs can be costly.
- Like in case of change of reference of an object.

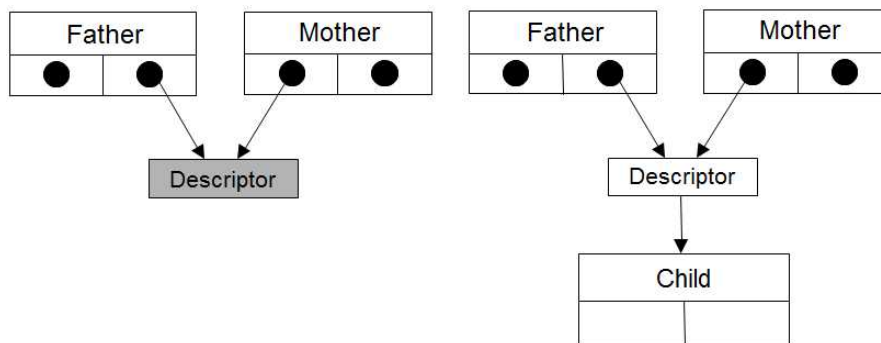


- Maintaining RRLs can be costly.
- Like in case of change of reference of an object.

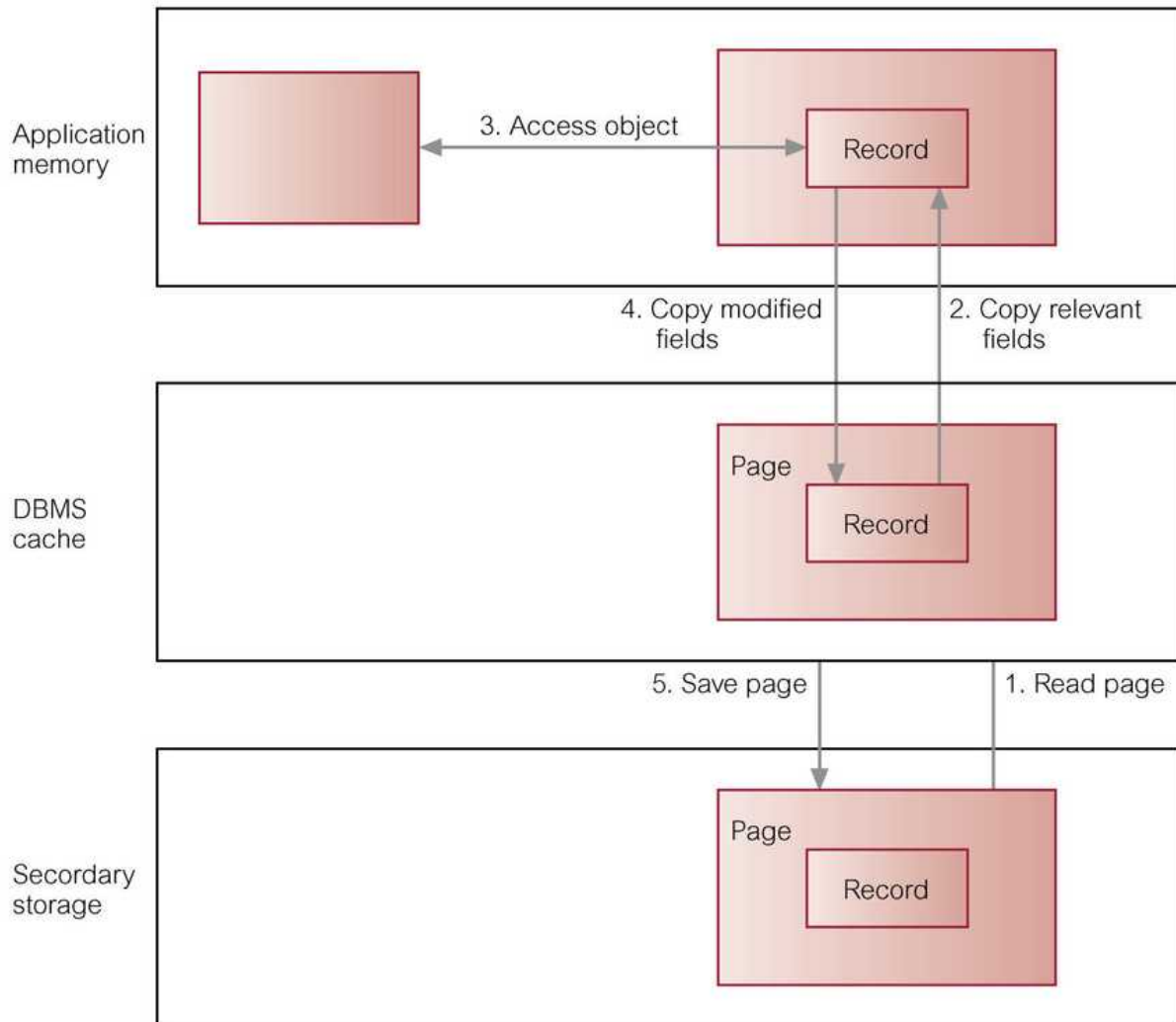


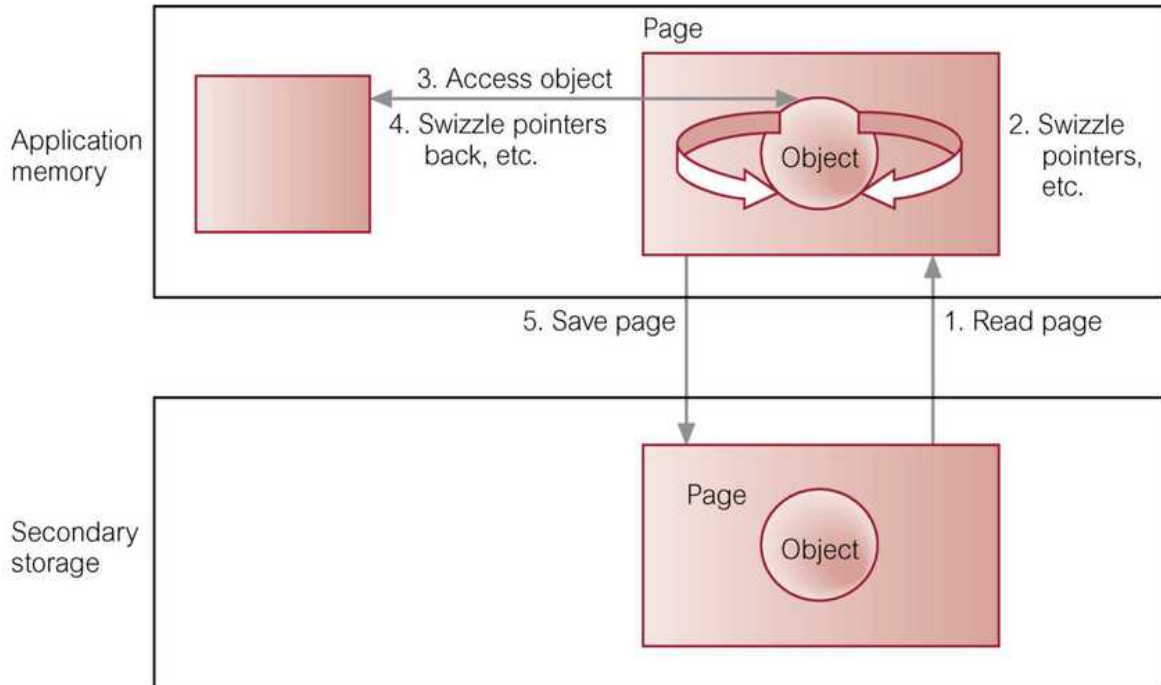
Indirect Swizzling

- With indirect swizzling, virtual memory pointer is placed in an intermediate object, which acts as a placeholder for the actual object.
 - ✓ If referenced object is resident then placeholder contains address otherwise NULL.



- Allows objects to be uncached without requiring swizzled pointers to be unswizzled.

Accessing an Object with RDBMS

Accessing an Object with OODBMS**Persistence**

- Transient v/s persistent objects.
- Initial approach was embedded language.

Persistent Schemes

- Consider three persistent schemes:
 - ✓ Checkpointing
 - ✓ Serialization
 - ✓ Explicit Paging
- **Note:** persistence can also be applied to (object) code and to the program execution state. 

Checkpointing

- Copy all or part of program's address space to secondary storage.
- If complete address space saved, program can restart from checkpoint.
- In other cases, only program's heap saved.

Serialization

- Copy closure of a data structure to disk.
- Write on a data value may involve traversal of graph of objects reachable from the value, and writing of flattened version of structure to disk.
- Sometimes called, pickling, or in a distributed computing context, marshaling.
- Two inherent problems:

- ✓ Does not preserve object identity.
- ✓ Not incremental, so saving small changes to a large data structure is not efficient.

Explicit Paging

- Explicitly 'page' objects between application heap and persistent store.
- Usually requires conversion of object pointers from disk-based scheme to memory-based scheme.
- Two common methods for creating/updating persistent objects:
 - ✓ Reachability-based
 - ✓ Allocation-based

Reachability-Based Persistence

- Object will persist if it is reachable from a persistent root object.
- Programmer does not need to decide at object creation time whether object should be persistent.
- Maps well onto language that contains garbage collection mechanism (e.g. Smalltalk or Java).

Allocation-Based Persistence

- Object only made persistent if it is explicitly declared as such within the application program. Can be achieved in several ways:
- By Class
 - ✓ Class is statically declared to be persistent and all instances made persistent when they are created.
 - ✓ Class may be subclass of system-supplied persistent class.
- By Explicit call
 - ✓ Object may be specified as persistent when it is created or dynamically at runtime.

Orthogonal Persistence

- Three fundamental principles:
 - ✓ Persistence independence.
 - ✓ Data type orthogonality.
 - ✓ Transitive persistence (originally referred to as 'persistence identification' but ODMG term 'transitive persistence' used here).

Persistence Independence

- Persistence of object independent of how program manipulates that object.
- Conversely, code fragment independent of persistence of data it manipulates.
- Should be possible to call function with its parameters sometimes objects with long term persistence and sometimes only transient.
- Programmer does not need to control movement of data between long-term and short-term storage.

Lecture -15**Previous Lecture**

- Pointer Swizzling
- Persistence
 - ✓ Checkpointing
 - ✓ Serialization
- Orthogonal Persistence
 - ✓ Explicit paging
 - ✓ Persistence Independence

Data Type Orthogonality

- All data objects should be allowed full range of persistence irrespective of their type.
- No special cases where object is not allowed to be long-lived or is not allowed to be transient.

Transitive Persistence

- Complex object persists along with its contained object.
- Issue: contained type itself persistent or not?

Questions on Orthogonal Persistence

- Objects in queries
- Persistent or transient or both?
- Objects in current user's area or other areas too?
- Indexes on transient objects too?
- Objects in transaction semantics?
 - ✓ ACID properties apply to persistent objects.
 - ✓ Should uncommitted changes be undone with the transient objects too.
 - ✓ That requires logging of transient objects updates too.

Issues in OODBMSs

- Long-duration Transactions
- Versions
- Schema Evolution

Transactions

- A logical unit of work.
- Must transform database from one consistent state to another one.
- Normal DB transactions are short lived.
- Complex objects involve long transactions.
- Such transactions need special CC mechanisms.
- Unit of CC is Object; CC normally implemented using locks.
- Aborting of a long transaction due to some conflict will not be acceptable to user.
- Two solutions
 - ✓ Multiversion CC protocols.
 - ✓ Advanced transaction models involving:
 - Nested transactions
 - Sagas

Multiversion CC

- Used in timestamp ordering.
- A technique to maintain serializability of transactions.
- Avoids deadlocks.
- New version created on write operation.
- May cause cancellation of transactions.

Nested Transactions

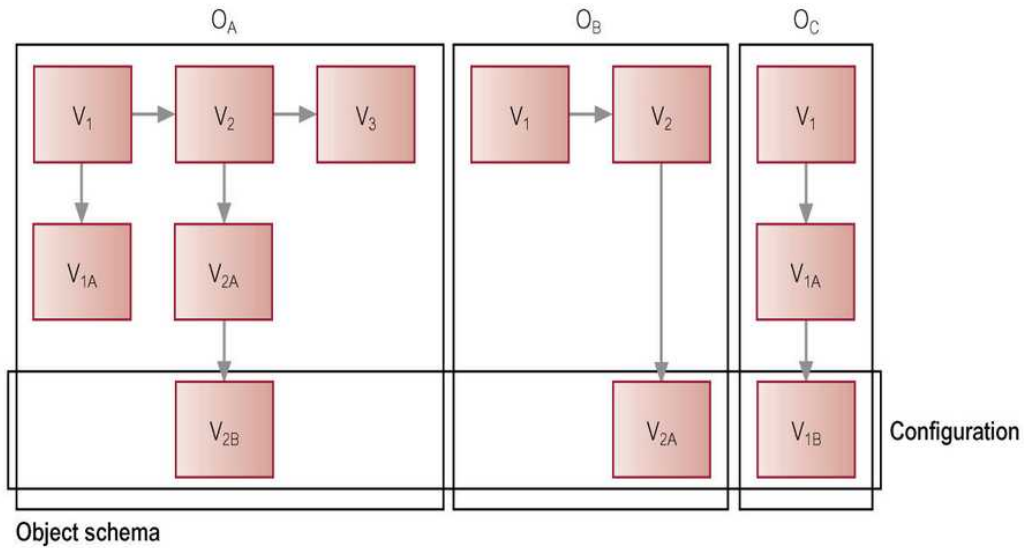
- Transaction viewed as hierarchy of sub-transactions.
- Each child can also have nested transactions.
- Transactions have to commit from bottom upwards.

begin_transaction T ₁	Complete Reservation
begin_transaction T ₂	Airline_reservation
begin_transaction T ₃	First_flight
reserve_airline_seat(London, Paris);	
commit T ₃ ;	
begin_transaction T ₄	Connecting_flight
reserve_airline_seat(Paris, New York);	
commit T ₄ ;	
commit T ₂ ;	
begin_transaction T ₅	Hotel_reservation
book_hotel(Hilton);	
commit T ₅ ;	
begin_transaction T ₆	Car_reservation
book_car();	
commit T ₆ ;	
commit T ₁ ;	

Sagas

- A sequence of (flat) transactions that can be interleaved with other transactions.
- DBMS guarantees that either all transactions in saga are successfully completed or compensating transactions are run to undo partial execution.

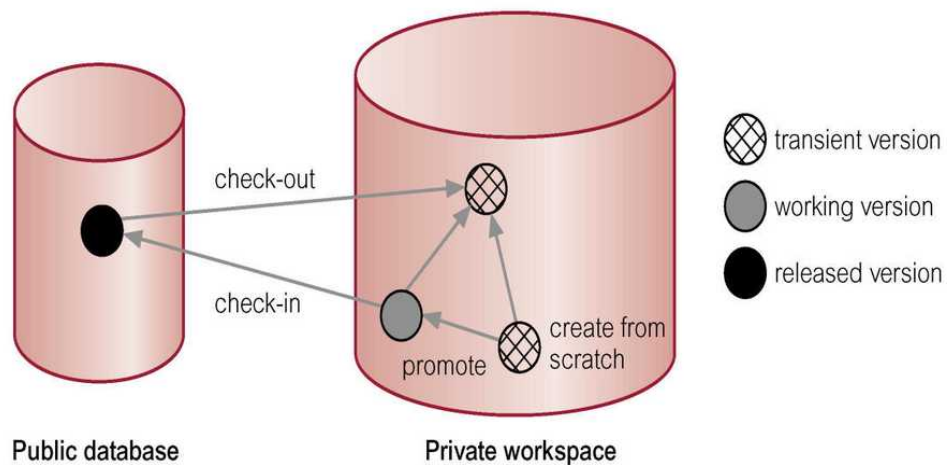
Versions and Configurations



ITASCA DBMS

- ITASCA identifies 3 types of versions:
 - ✓ Transient Versions
 - ✓ Working Versions
 - ✓ Released Versions
- Versionable or non-versionable object.

Versions and Configurations

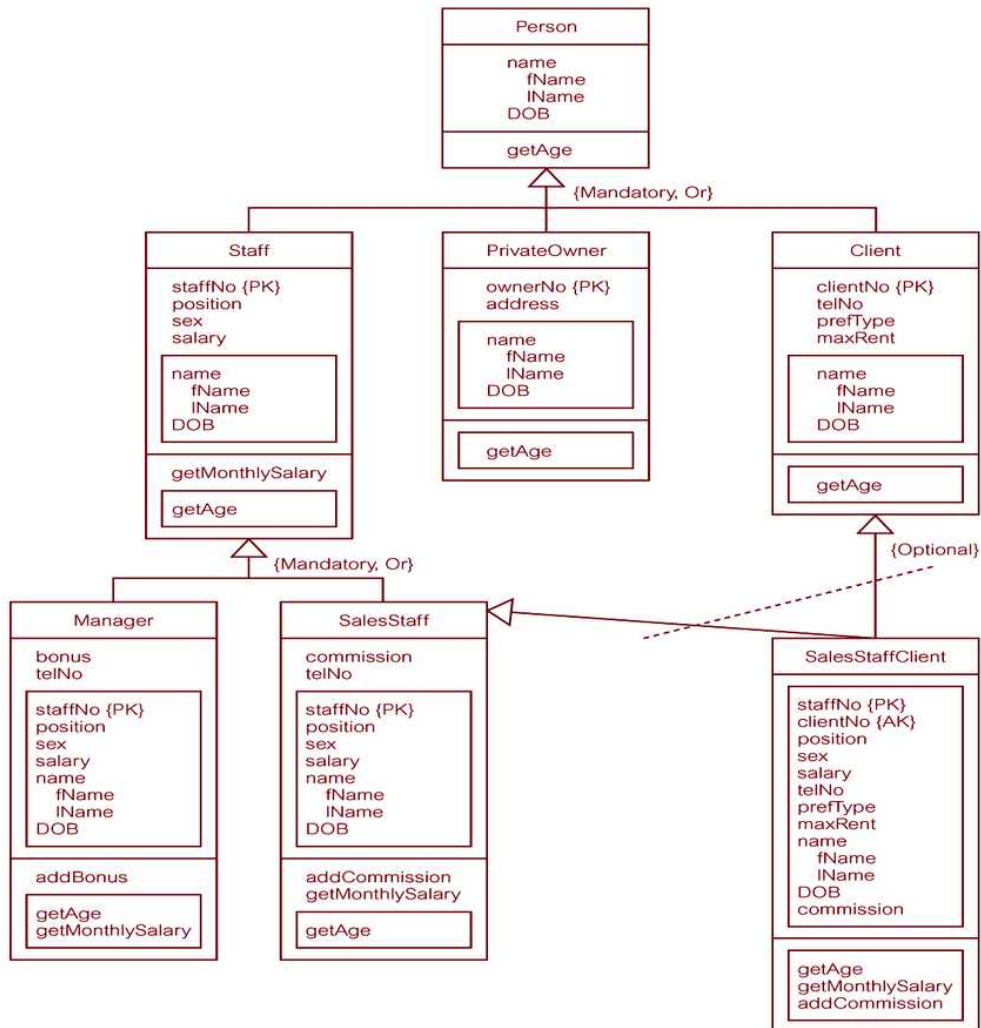


Schema Evolution

- And after system deployment, like:
 - ✓ Modifying class definition
 - ✓ Inheritance structure
- Typical schema changes:
 1. Changes to class definition:
 - a) Modifying Attributes
 - b) Modifying Methods
 2. Changes to inheritance hierarchy:
 - a) Making a class S superclass of a class C
 - b) Removing S from list of superclasses of C
 - c) Modifying order of superclasses of C
 3. Changes to set of classes, such as creating and deleting classes and modifying class names.
- Changes must not leave schema inconsistent.

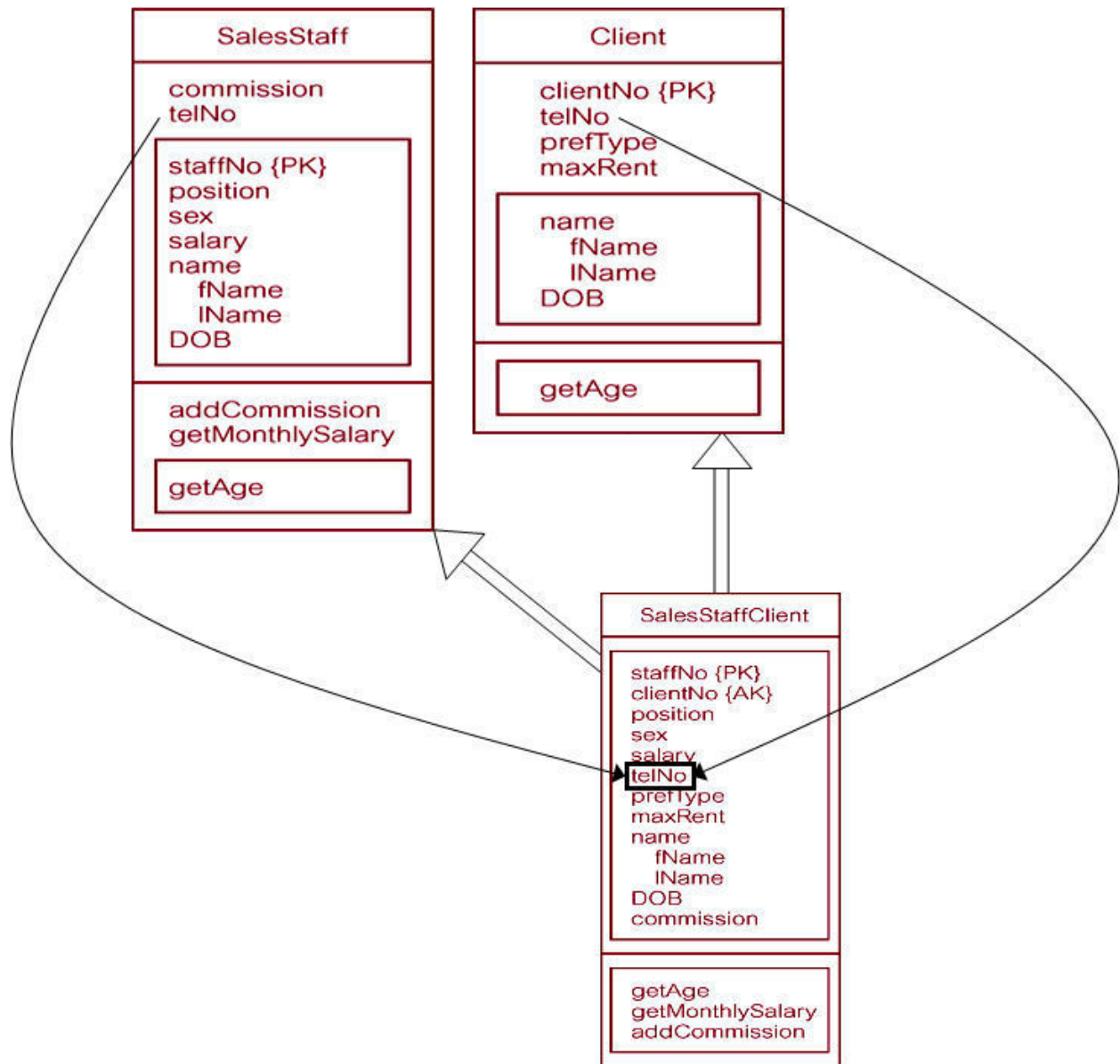
Lecture -16

Schema Consistency

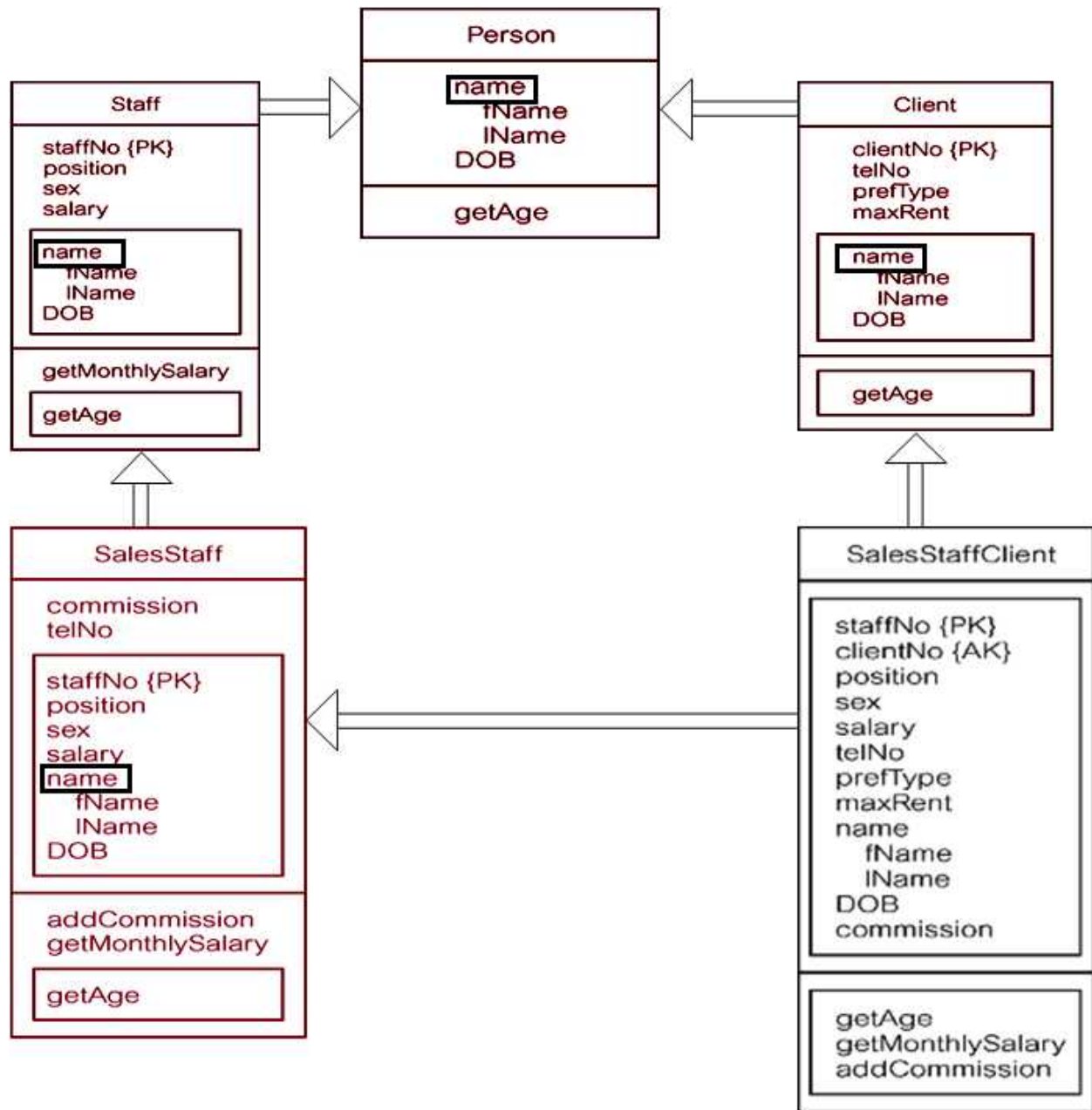


1. Resolution of conflicts caused by multiple inheritance and redefinition of attributes and methods in a subclass.
 - 1.1 Rule of precedence of subclasses over superclasses.
 - 1.2 Rule of precedence between superclasses of a different origin (`telNo`).

Rule of precedence between superclasses of a different origin



1.3 Rule of precedence between superclasses of the same origin (name).

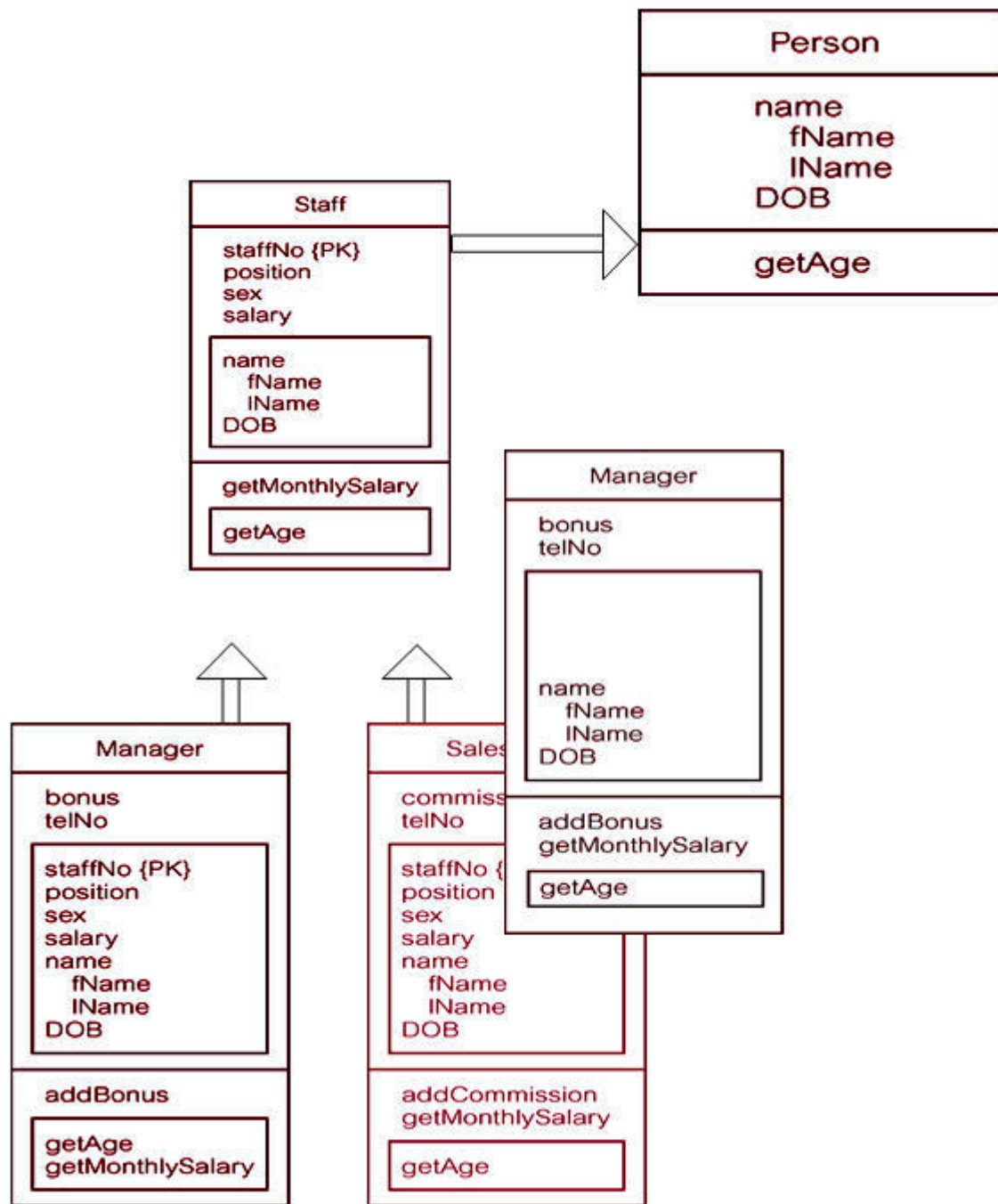


2. Propagation of modifications to subclasses.

- 2.1 Rule for propagation of modifications (e.g. delete).
- 2.2 Rule for propagation of modifications in the event of conflicts (when there is no name conflict).
- 2.3 Rule for modification of domains (can be generalized, not in subclass).

3. Aggregation and deletion of inheritance relationships between classes and creation and removal of classes.

- 3.1 Rule for inserting super classes.
- 3.2 Rule for removing super classes (Staff).



3.3 Rule for inserting a class into a schema (No superclass).

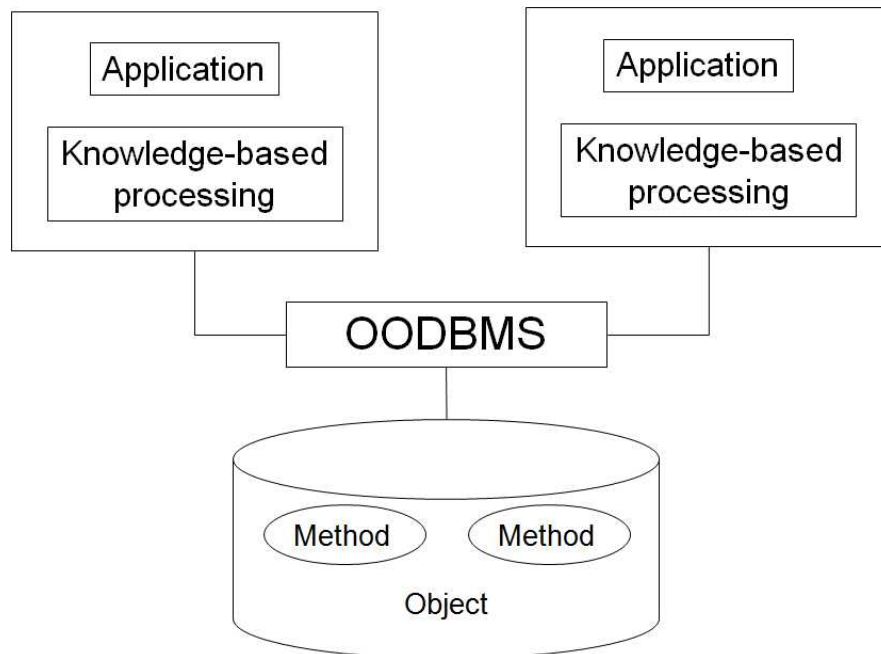
3.4 Rule for removing a class from a schema.

OODBMS Architectures

- OO + DBMS = OODBMS

Architecture

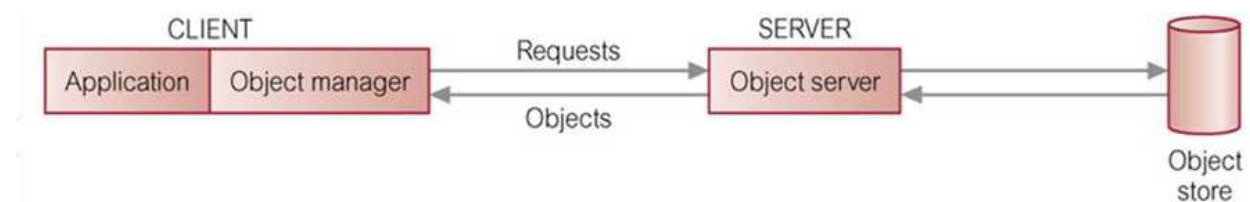
- The basic theme of OODBMSs is to add persistence to OOPL as they provide object orientation.
- The major difference is that here we need to store data as well as methods.
- Data of arbitrary types.

OODBMS Architecture**Client-Server Architecture**

- Three basic architectures:
 - ✓ Object Server
 - ✓ Page Server
 - ✓ Database Server

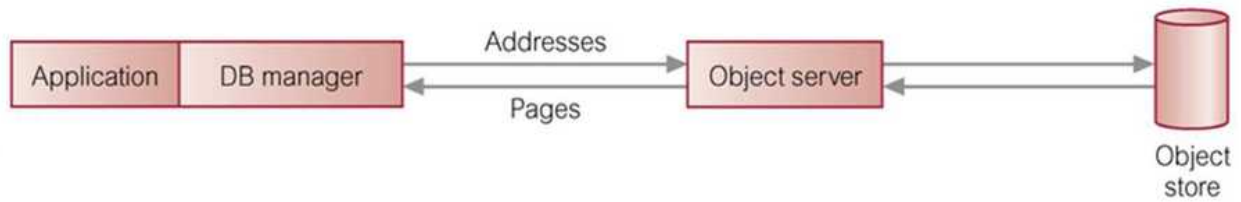
Object Server

- Distribute processing between the two components.
- Typically, Server responsible for other DBMS functions.
- Client is responsible for transaction management and interfacing to programming language.
- Best for cooperative, object-to-object processing in an open, distributed environment.

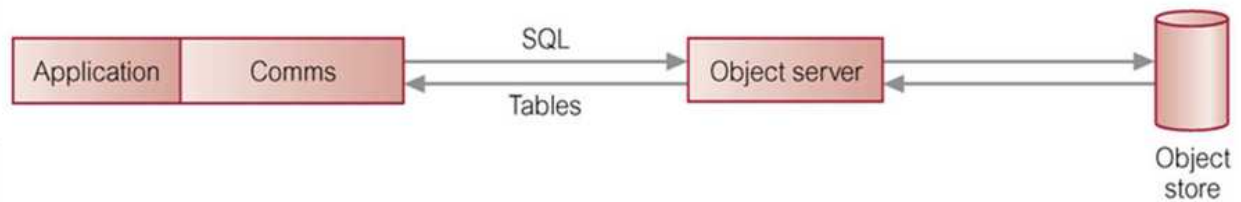


Page Server

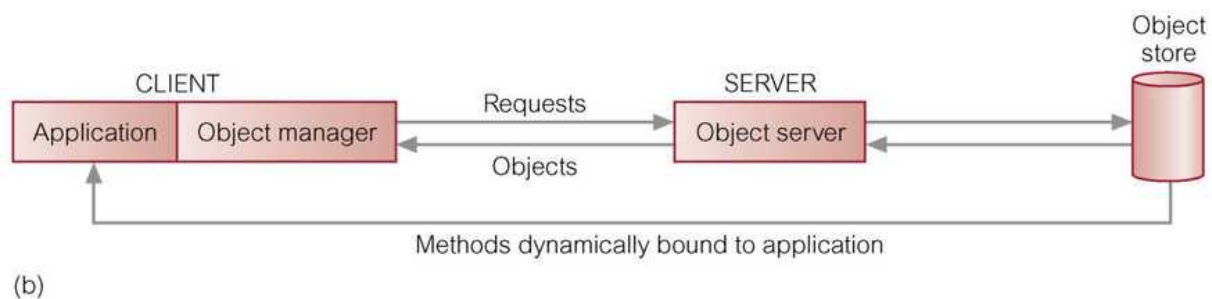
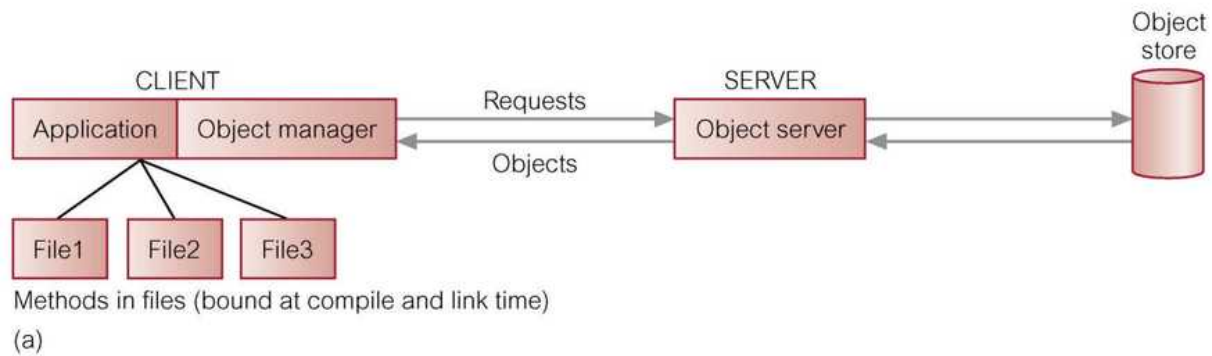
- Most database processing is performed by client.
- Server responsible for secondary storage and providing pages at client's request.

**Database Server**

- Most database processing performed by server.
- Client simply passes requests to server, receives results and passes them to application.
- Approach taken by many RDBMSs.

**Storing and Executing Methods**

- Two approaches:
 - ✓ Store methods in external files.
 - ✓ Store methods in database.



- Benefits of latter approach:
 - ✓ Eliminates redundant code
 - ✓ Simplifies modifications
 - ✓ Methods are more secure
 - ✓ Methods can be shared concurrently
 - ✓ Improved integrity
- Obviously, more difficult to implement.

Benchmarking

- A general term used in different areas, like business.
- It is the process of comparing one's business processes and performance metrics to industry bests and/or best practices from other industries [wiki].
- Objective is to measure the performance of any thing; typically time, cost & quality.
- System specific BM defined.
- One initially used for RDBMSs is Wisconsin Benchmark.

Benchmarking - Wisconsin benchmark

- Original benchmark had 3 relations:
 - ✓ Onetup (1000 tuples)
 - ✓ Tentup1 (10000 tuples)
 - ✓ Tentup2 (10000 tuples)
- Consortium of manufacturers formed Transaction Processing Council (TPC) in 1988 to create series of transaction-based test suites to measure database/TP environments.

TPC Benchmarks

- TPC-A and TPC-B for OLTP (now obsolete).
- TPC-C replaced TPC-A/B and based on order entry application.
- TPC-H for ad hoc, decision support environments.
- TPC-R for business reporting within decision support environments.

Object Operation Version 1 (OO1) Benchmark

- De-facto standards are OO1 (Cattell BM), HyperModel, OO7.

OO1 (Cattell BM)

- Intended as generic measure of OODBMS performance.
- Designed to reproduce operations common in advanced engineering applications, such as finding all parts connected to a random part, all parts connected to one of those parts, and so on, to a depth of seven levels.

Object Operation Version 1 (OO1) Benchmark

- About 1990, benchmark was run on GemStone, Ontos, ObjectStore, Objectivity/DB, and Versant, and INGRES and Sybase. Results showed an average 30-fold performance improvement for OODBMSs over RDBMSs.

HyperModel BM (1990)

- Possesses a richer workload model than OO.
- Potentially more effective than OO1 in measuring the performance of engineering databases.

OO7 Benchmark**Comparison of existing benchmarks**

	Relevance	Portability	Simplicity	Scalability
OO1	--	++	++	-
HyperModel	+	+	-	--
OO7	++	+	--	-
OCB	++	+	-	++

Strong point: + Very strong point: ++ Weak point: - Very weak point: --

Lecture -17**Previous Parts**

- Previous Part
 - ✓ OODBMS Perspective
 - ✓ Issues in OODBMS
- Earlier in the Course
 - ✓ Problems in DBMSs
 - ✓ The need for OODBMS
 - ✓ Extended Relational DBMS
- Earlier in the Course
 - ✓ Object Relational Systems
 - ✓ Pure OODBMSs

Part-5

- ODBMSs Standards
 - ✓ Object Data Management Group (ODMG)
 - ✓ Object Management Group (OMG)

Object Data Management Group – (ODMG)

- References
 - ✓ R. Cattell, D. Barry, “The Object Data Standard: ODMG 3.0”, Morgan Kaufman Publishers
 - ✓ T. Connolly, C. Begg, “Database System: A Practical Approach to Design, Implementation and Management” Fourth Edition, Ch 27
- Established in 1991, originally Object Database Management Group.
- Founder Rick Cattell of Sun Microsystems.

ODMG – Milestones

- Established 1991
- ODMG 1.0 in 1993
- ODMG 2.0 1997
- ODMG 3.0 1999-2001

Components of ODMG 3.0

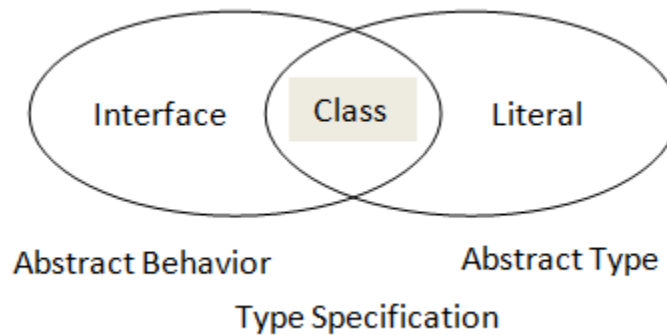
- Object Model (OM)
- Object Specification Languages
 - (mainly ODL & Interface Language)
- Object Query Language (OQL)
- Language Binding
 - ✓ C++
 - ✓ Smalltalk
 - ✓ Java

ODMG 3.0

- OM of ODMG 3.0
- Reference: Chapter 2 of ODMG 3.0
- Basic building blocks
 - ✓ Object and Literal.
 - ✓ Types-having common range of states and behavior.
 - ✓ State characterized by properties that include attributes and relationships.
- Basic building blocks
 - ✓ Behavior by operations executed on objects.
 - ✓ ODMS stores objects defined in ODL.

Specifications and Implementation

- Type has an external spec and one or more implementations.
- Specification: external characteristics of type; visible to users.
 - ✓ Operations, state variables and exceptions.
- Implementation: internal detail of objects of type; determined by a language binding.
- Interface defines abstract behavior of an object type.
- **Class definition gives abstract state and abstract behavior of object type.**
- Literal.



- `interface Employee {...};` `// abstract behavior`
- `class Person {...};` `// abstract state and behavior`
- `struct Complex {float re; float im};` `// abstract state`
- Data structure and methods depend on language binding.

Subtyping and Inheritance of Behavior

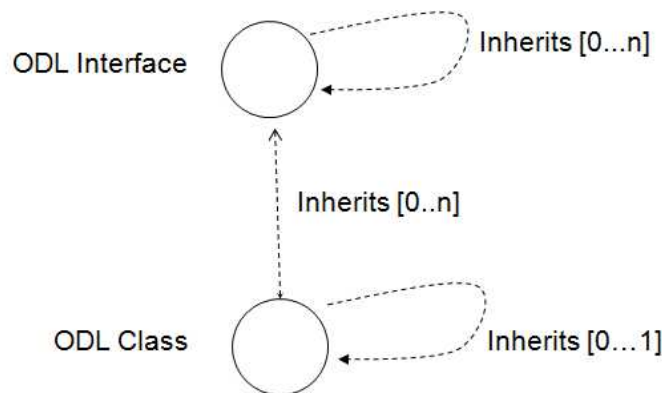
- OM supports inheritance of behavior; represented in graphs.
- *Interface Employee {...};*
- *Interface Prof: Employee {...};*
- *Interface A_Prof: Prof {...};*
- ```
Interface Employee {
 void getEmp();
 int hireEmp(in E Emp) raises (inSufData);
 void showEmp();
};
```
- ```
Interface Prof: Employee {
    void    listPubs() raises (noPubsProvided);
    string  enterVisit(in C Country);
};
```
- *Interface A_Prof: Prof {...};*

Lecture -18

ODMG 3.0

Specification & Implementation

- Classes can inherit from interfaces.
- *class salEmp: Employee {...};*
- *class horlyEmp: Employee {...};*
- Multiple inheritance of behavior supported; operations can be refined in subclasses.
- ODL classes are mapped to PL classes by language binding.
- Classes can be instantiated but interfaces can't be.



Inheritance of State

- ISA defines inh of behavior, EXTENDS defines of state and behavior; only to classes not to literals; single inheritance.
- EXTENDS is transitive.
- ```
class Person {
 attribute string name;
 attribute Date birthDate;
};
```
- ```
class EmPerson extends Person: Employee {
    attribute Date hireDate;
    relationship Manager boss inverse
        Manager::subordinates;
};
```
- ```
class MangPerson extends EmPerson : Manager {
 relationship set<Employee> subordinates
 inverse Employee::boss;
};
```
- Duplication of declarations in Interface and Class; like of boss and subordinates; help to access the relevant objects.

**Integrity and State****Extents**

- Set of all instances of a type.
- In ODMG, schema designer decides whether to maintain an explicit extent for a type.
- Includes creating, inserting, removing instances.

**ODMG Objects**

- Creation
- Identifiers
- Names
- Lifetime
- Structure

**Enclosing Module**

- module ODLTypes {
  - exception DatabaseClosed{ };
  - exception TransactionInProgress { };
  - exception TransactionNotInProgress{ };
  - exception IntegrityError{ };
  - exception LockNotGranted{ };
- };

**Object Creation**

- Invoking creation operation
  - interface objectFactory {
    - Object new();
  - };
- Defined on Factory object provided by binding.

**ODL Interface for All Objects**

```

interface ObjectFactory {
 Object new();
}
interface Object {
 enum Lock_Type{read, write, upgrade};
 void lock(in Lock_Type mode) raises(LockNotGranted); // obtain lock – wait if necessary
 boolean try_lock(in Lock_Type mode); // obtain lock – do not wait if not immediately granted
 boolean same_as(in Object anObject); // identity comparison
 Object copy(); // copy object – copied object not ‘same as’
 void delete(); // delete object from database
};

```

**Object Identifiers**

- OID is unique in storage domain; ODMS here.
- Object retains same OID in its lifetime; value can change.
- Different from PK of RDBMS.

**Lecture -19****Object Identifiers/Object Names****Object Names**

- Can also have one or more names.
- ODMS function relates names with OID.
- Scope of uniqueness is ODMS.

**Literals**

- A constant, with possibly complex structure.
- Literal types decomposed as atomic, collections, structured, or null.
- Values of a literal's properties may not change.
- Do not have their own identifiers and cannot stand alone as objects.
- Embedded in objects and cannot be individually referenced.
- Structured literals contain fixed number of named heterogeneous elements.

**Lifetime**

- Determines the memory management for object:
  - ✓ Transient
  - ✓ Persistent
- Transient in memory; managed by PL runtime system.
- Transient declared in a procedure or scoped with the process.
- Persistent assigned memory and storage by ODMS; persist even if process is terminated.

**Structure of Objects**

- Atomic objects: user-defined types; no built in atomic object types.
- Collection objects: consist of distinct elements of atomic, collection or literal.

**Interface of Collection**

|                                |                                      |                                    |
|--------------------------------|--------------------------------------|------------------------------------|
| interface Collection: Object { |                                      |                                    |
| exception                      |                                      | InvalidCollection{};               |
| exception                      |                                      | ElementNotFound{Object element};   |
| unsigned long                  | cardinality();                       | // return number of elements       |
| boolean                        | is_empty();                          | // check if collection empty       |
| boolean                        | is_ordered();                        | // check if collection is ordered  |
| boolean                        | allows_duplicates();                 | // check if duplicates are allowed |
| boolean                        | contains_element(in Object element); | // check for specified element     |
| void                           | insert_element(in Object element);   | // insert specified element        |

```

void remove_element(in Object element) // remove specified element
 raises(ElementNotFound);
Iterator create_iterator(in boolean stable);
 // create forward only traversal iterator

BidirectionalIterator create_bidirectional_iterator(in boolean stable)
 raises(InvalidCollectionType); // create bi-directional iterator
Object select_element(in string OQL-predicate);
Iterator select(in string OQL-predicate);
boolean query(in string OQL-predicate, inout Collection result);
boolean exists_element(in string OQL-predicate); };

```

**Iterator**

- Iterator: a mechanism to traverse a collection.
- Its interface has been defined that contains different operations.

**Iterator – Interface**

```

interface iterator {
 exception NoMoreElements{ };
 exception InvalidCOLlectionType{ };
 boolean is_stable(); // modifications in collection during iteration doesn't effect traversal
 boolean at_end();
 void reset();
 Object get_element() raises (NoMoreElements);
 void next_position() raises (NoMoreElements);
 void replace_element(in Object element)
 raises (InvalidCOLlectionType);
};

```

```

Interface BidirectionalIterator: Iterator {
 boolean at_begining();
 void previous_position()
 raises (NoMoreElement)
};

```

**Set Objects**

- Unordered collection of elements.
  - No duplication allowed.
- ```

interface SetFactory: ObjectFactory {
    Set        new_of_size(in long size);
};

```

Set Objects – Interface

```

class Set : Collection {
    attribute      set<t> value;
    Set            create_union(in Set other_set);
    Set            create_intersection(in Set other_set);
    Set            create_difference(in Set other_set);
    boolean        is_subset_of(in Set other_set);
    boolean        is_proper_subset_of(in Set other_set);
    boolean        is_superset_of(in Set other_set);
    boolean        is_proper_superset_of(in Set other_set);
};

```

Bag Objects

- Unordered collection of elements.
 - Can contain duplicates.
- ```

interface BagFactory: ObjectFactory {
 Bag new_of_size(in long size);
};

```

**Bag Objects – Interface**

```

class Bag : Collection {
 attribute bag<t> value;
 unsigned long occurrences_of (in Object element);
 Bag create_union(in Bag other_bag);
 Bag create_intersection(in Bag other_bag);
 Bag create_difference(in Bag other_bag);
};

```

**Lecture -20****List Objects**

- Ordered collection of elements.
- Operations are positional in nature in ref to a given index or start or end of List object.
- Index starts from zero (0).
- interface ListFactory: ObjectFactory {  
     List     new\_of\_size(in long size);  
 };

**List Interface**

```
class List: Collection {
 exception InvalidIndex {undigned long index; };
 attribute list<t> value;
 void remove_element_at(in unsigned long index) raises(InvalidIndex);
 Object retrieive_element_at(in unsigned long index)raises(InvalidIndex);
 void replace_element_at(in Object element, in unsigned long index) raises(InvalidIndex);
 void insert_element_after(in Object element, in unsigned long index) raises(InvalidIndex);
 void insert_element_before(in Object element, in unsigned long index) raises(InvalidIndex);
 void insert_element_first(in Object element)
 void insert_element_last(in Object element)
 void remove_first_element() raises (ElementNotFound);
 void remove_last_element() raises (ElementNotFound);
 Object retrieve_first_element() raises (ElementNotFound);
 Object retrieve_last_element() raises (ElementNotFound);
 List concat(in List other_list);
 void append(in List other_list);
};
```

**Array Objects**

- A dynamically sized, ordered collection of elements.
- Can be located by position.
- Index starts from zero (0).

**Array Factory & Interface**

```
interface ArrayFactory : ObjectFactory {
 Array new_of_size(in long size);
};
```

```
class Array : Collection {
 exception InvalidIndex{unsigned long index; };
 exception InvalidSize{unsigned long size; };
 attribute array<t> value;
```

```

void replace_element_at(in unsigned long index, in Object element) raises(InvalidIndex);
void remove_element_at(in unsigned long index)raises(InvalidIndex);
Object retrieve_element_at(in unsigned long index)raises(InvalidIndex);
void resize(in unsigned long new_size)raises(InvalidSize);
};

```

### Dictionary Objects

- A Dictionary object is an unordered sequence of key-value pairs with no duplicate keys.
- Each key-value pair is constructed as an instance of the following structure:  

```
struct Association
{Object key; Object value; };
```
- Iterating over a Dictionary object will result in the iteration over a sequence of Associations.
- Each `get_element` operation, executed on an Iterator object, returns a structure of type Association.

### Dictionary Factory & Interface

```

interface DictionaryFactory : ObjectFactory {
 Dictionary new_of_size(in long size);
};

class Dictionary : Collection {
 exception DuplicateName{string key; };
 exception KeyNotFound{Object key; };
 attribute dictionary<t,v>value;
 void bind(in Object key, in Object value) raises(DuplicateName);
 void unbind(in Object key) raises(KeyNotFound);
 Object lookup(in Object key) raises(KeyNotFound);
 boolean contains_key(in Object key);
};

```

### Collection Objects

#### Structured Objects

- All structured objects support the Object ODL interface.
  - ✓
  - ✓ Date
  - ✓ Interval
  - ✓ Time
  - ✓ Timestamp

**Structured Objects Date**

```

interface DateFactory : ObjectFactory {
 exception InvalidDate{};
 Date julian_date(in unsigned short year, in unsigned short julian_day)
 raises(InvalidDate);
 Date calendar_date(in unsigned short year, in unsigned short month, in unsigned
 short day) raises(InvalidDate);
 boolean is_leap_year(in unsigned short year);
 boolean is_valid_date(in unsigned short year, in unsigned short month, in unsigned
 short day);
 unsigned short days_in_year(in unsigned short year);
 unsigned short days_in_month(in unsigned short year, in Date::Month month);
 Date current();
};

class Date : Object {
 enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
 enum Month {January, February, March, April, May, June, July, August, September, October,
 November, December};
 attribute date value;
 unsigned short year();
 unsigned short month();
 unsigned short day();
 unsigned short day_of_year();
 Month month_of_year();
 Weekday day_of_week();
 boolean is_leap_year();
 boolean is_equal(in Date a_date);
 boolean is_greater(in Date a_date);
 boolean is_greater_or_equal(in Date a_date);
 boolean is_less(in Date a_date);
 boolean is_less_or_equal(in Date a_date);
 boolean is_between(in Date a_date, in Date b_date);
 Date next(in Weekday day);
 Date previous(in Weekday day);
 Date add_days(in long days);
 Date subtract_days(in long days);
 long subtract_date(in Date a_date);
};

```

**Time**

- Denotes specific world times, which are internally stored in Greenwich Mean Time (GMT).
- Time zones are specified according to the number of hours that must be added or subtracted from local time.

**Timestamp**

- Timestamps consist of an encapsulated Date and Time.
- interface TimestampFactory : ObjectFactory {
 

```

 exception InvalidTimestamp{Date a_date, Time a_time; };
 Timestamp current();
 Timestamp create(in Date a_date, in Time a_time)
 raises(InvalidTimestamp);
 };

```
- class Timestamp : Object {
 

```

 attribute timestamp value;
 Date get_date();
 Time get_time();
 unsigned short year();
 unsigned short month();
 unsigned short day();
 unsigned short hour();
 unsigned short minute();
 unsigned short second();
 unsigned short millisecond();
 short tz_hour();
 short tz_minute();
 Timestamp plus(in Interval an_interval);
 Timestamp minus(in Interval an_interval);
 boolean is_equal(in Timestamp a_stamp);
 boolean is_greater(in Timestamp a_stamp);
 boolean is_greater_or_equal(in Timestamp a_stamp);
 boolean is_less(in Timestamp a_stamp);
 boolean is_less_or_equal(in Timestamp a_stamp);
 boolean is_between(in Timestamp a_stamp, in Timestamp b_stamp);
 };

```

**Interval**

- Intervals represent a duration of time and are used to perform some operations on Time and Timestamp objects.
- Created using the subtract\_time operation defined in the Time interface.

- class Interval : Object {
  - attribute interval value;
  - unsigned short day();
  - unsigned short hour();
  - boolean is\_zero();
  - Interval plus(in Interval an\_interval);
  - Interval minus(in Interval an\_interval);
  - Interval product(in long val);
  - Interval quotient(in long val);
  - boolean is\_equal(in Interval an\_interval);
  - boolean is\_greater(in Interval an\_interval);
  - boolean is\_greater\_or\_equal(in Interval an\_interval);
  - boolean is\_less(in Interval an\_interval);
  - boolean is\_less\_or\_equal(in Interval an\_interval);
- };

### Literal Types

- Value that doesn't change; like constant.
- ODMG supports:
  - ✓ Atomic literal
  - ✓ Collection literal
  - ✓ Structured literal

### Atomic Literal

- Atomic Literals: single or simple values, like 23.45 , "Object- Oriented Database Management Systems".
- Instances created implicitly, not by application program.
- ODMG supported literal types are:
- ODMG supported literal types:
 

|                  |                      |
|------------------|----------------------|
| ✓ long           | ✓ double             |
| ✓ long long      | ✓ boolean            |
| ✓ short          | ✓ octet              |
| ✓ unsigned long  | ✓ char (character)   |
| ✓ unsigned short | ✓ string             |
| ✓ float          | ✓ enum (enumeration) |

**Lecture -21****Collection Literals & Structured Literals****Collection Literals**

- set<t>
- bag<t>
- list<t>
- array<t>
- dictionary<t,v>
- Same as collection objects; except OID.

**Structured Literals**

- Structure has a fixed number of elements.
- Each element has a variable name.
- Can contain either a literal value or an object.
- Built-in: date, time, interval, timestamp and also user-defined.
- Copy and comparison allowed on the basis value, not ID.
- Equivalence of literals.
- If they have the same literal type and are both atomic and contain the same value.
  - ✓ If both are sets, have the same parameter type t.
  - ✓ If t is a literal type, then for each element in x, there is an element in y that is equivalent to it, and, for each element in y, there is an element in x that is equivalent to it.
  - ✓ If t is an Object type, then both x and y contain the same set of object identifiers.
  - ✓ Are both bags, have the same parameter type t.
  - ✓ if t is a literal type, then for each element in x, there is an element in y that is equivalent to it, and, for each element in y, there is an element in x that is equivalent to it. In addition, for each literal appearing more than once in x, there is an equivalent literal occurring the same number of times in y.
  - ✓ if t is an Object type, then both x and y contain the same set of object identifiers. In addition, for each object identifier appearing more than once in x, there is an identical object identifier appearing the same number of times in y.
  - ✓ Are both arrays or lists, have the same parameter type t, and for each entry i.
  - ✓ If t is a literal type, then x[i] is equivalent to y[i] (equal).
  - ✓ If t is an object type, then x[i] is identical to y[i] (same\_as).

**ODMG Supported Type Hierarchy****Literal\_type**

- Atomic\_literal
  - ✓ Long
  - ✓ Long long
  - ✓ Short
  - ✓ Unsigned long
  - ✓ Unsigned short
  - ✓ Float
  - ✓ Double
  - ✓ Boolean
  - ✓ Octet
  - ✓ Char
  - ✓ String
  - ✓ Enum <> // enumeration

- Collection\_literal
    - ✓ Set<>
    - ✓ bag<>
    - ✓ list<>
  - Structured\_literal
    - ✓ Date time
    - ✓ timestamp
- ✓ array<>
  - ✓ Dictionary<>
  - ✓ interval
  - ✓ structure<>

### object\_type

- Atomic\_object
  - Collection\_object
    - ✓ Set<>
    - ✓ Bag<>
    - ✓ List<>
  - Structured\_object
    - ✓ Date
    - ✓ Time
- ✓ Array<>
  - ✓ Dictionary<>
  - ✓ Timestamp
  - ✓ Interval

### ODMG Object Model

- The ODMG Object Model is strongly typed.
- Type equivalence based on declaration.
- Type compatibility follows subtyping defined in type hierarchy; so TS object can be assigned to variable of type T.
- No implicit conversion provided.
- Implicit type conversion may be provided by PL; not for the structured literals.

### Modeling State - Properties

- Class defines properties representing state using:
  - ✓ Attributes – have one type
  - ✓ Relationships – defined bw 2 types
- Attribute declaration defines abstract state of class; for example:
- class Person {
 

attribute short age;
 attribute string name;
 attribute enum gender {male, female};
 attribute Address home\_address;
 attribute set<Phone\_no> phones;
 attribute Department dept;

 };

**Modeling State - Relationship**

- Relationships: defined between two types.
- Only binary supported; cardinality.
- Defined explicitly by declaration of traversal paths; applications use this logical connection to access objects.
- class Professor {  
...  
relationship set<Course> teaches  
inverse Course::is\_taught\_by;  
...  
}
- And inversely...
- class Course {  
...  
relationship Professor is\_taught\_by  
inverse Professor::teaches;  
..  
}
- Different from declaring two attributes in two classes of reverse type.
- Different from pointers too.

**Relationship – Interface**

- Implementation of Many Cardinality.

```
relationship set<X> Y inverse Z;
readonly attribute set<X> Y;
void form_Y(in X target) raises(IntegrityError);
void drop_Y(in X target) raises(IntegrityError);
void add_Y(in X target) raises(IntegrityError);
void remove_Y(in X target) raises(IntegrityError);
```

```
relationship set<Course> teaches inverse course::is_taught_by;
readonly attribute set<Course> teaches;
void form_teaches(in Course aCourse)raises(IntegrityError);
void drop_teaches(in Course aCourse)raises(IntegrityError);
void add_teaches(in Course aCourse)raises(IntegrityError);
void remove_teaches(in Course aCourse)raises(IntegrityError);
```

**Modeling Behavior**

- Specified by a set of operations' signatures that include:
  - ✓ Name of operation
  - ✓ Name and type of arguments
  - ✓ Types of the value(s) returned
  - ✓ Any exceptions
- Operation defined on only single type.
- Overloading of names allowed but needs to be resolved.
- Model supports dynamically nested exception handlers.

**Exceptions**

- An exception handler declaration within a scope.
- An operation may "raise" an exception.
- The exception is "caught" by an exception handler.
- An exception handler capable of handling exceptions of type t will also handle exceptions of any subtype of t.
- However exceptions of a specific subtype of t may also be declared.

**Locking and Concurrency Control**

**Lecture -22****Database Consistency****Concurrency Issues**

- Lost Update Problem.
- Un-Committed Update Problem.
- Inconsistent analysis problem.

**Concurrency Issues – Causes**

- Two transactions accessing the same data item, with the condition that either one of them or both transactions are writing a particular data item.

**Concurrency Issues**

- ODMS supports serializability by monitoring compatibility of requested locks requests.

**Increasing Concurrency Level**

- The level of concurrency can be increased using:
  - ✓ Lock Compatibility
  - ✓ Granularity Level

**Deadlocks in Transactions****Locking Mechanism**

- Three Lock Types
  - ✓ Read (shared access)
  - ✓ Write (Exclusive access)
  - ✓ Upgrade

**Good Luck for Mid-Term Exam** 

**Lecture -23****Locking Mechanism****Transaction Model**

- Implicit and explicit locks supported (lock & try\_lock).
- Supports:
  - ✓ ACID properties
  - ✓ Serializability

**Transaction Operations**

- In two interfaces; one for creation and one for other operations.
- interface TransactionFactory {
 

```

 Transaction new();
 Transaction current();

```
- interface Transaction {
 

```

 void begin() raises(TransactionInProgress, DatabaseClosed);
 void commit() raises(TransactionNotInProgress);
 void abort() raises(TransactionNotInProgress);
 void checkpoint() raises(TransactionNotInProgress);
 void join() raises(TransactionNotInProgress);
 void leave() raises(TransactionNotInProgress);
 boolean isOpen();

```

**Database Operations**

- An ODMS may manage one or more logical ODMSs, each of which may be stored in one or more physical persistent stores.
- Each logical ODMS is an instance of the type Database; ODMS supplies.
- interface DatabaseFactory {
 

```

 Database new();

```
- interface Database {
 

```

 exception DatabaseOpen{};
 exception DatabaseNotFound{};
 exception ObjectNameNotUnique{};
 exception ObjectNameNotFound{};
 void open(in string odms_name) raises(DatabaseNotFound, DatabaseOpen);
 void close() raises(DatabaseClosed, TransactionInProgress);
 void bind(in Object an_object, in string name)
 raises(DatabaseClosed, objectNameNotUnique, TransactionNotInProgress);

```

```

Object unbind(in string name)
 raises(DatabaseClosed, ObjectNameNotFound, TransactionNotInProgress);
Object lookup(in string object_name)
 raises(DatabaseClosed, ObjectNameNotFound, TransactionNotInProgress);
ODLMetaObjects::Module schema()
 raises(DatabaseClosed, TransactionNotInProgress);
};

```

### Object Specification Languages

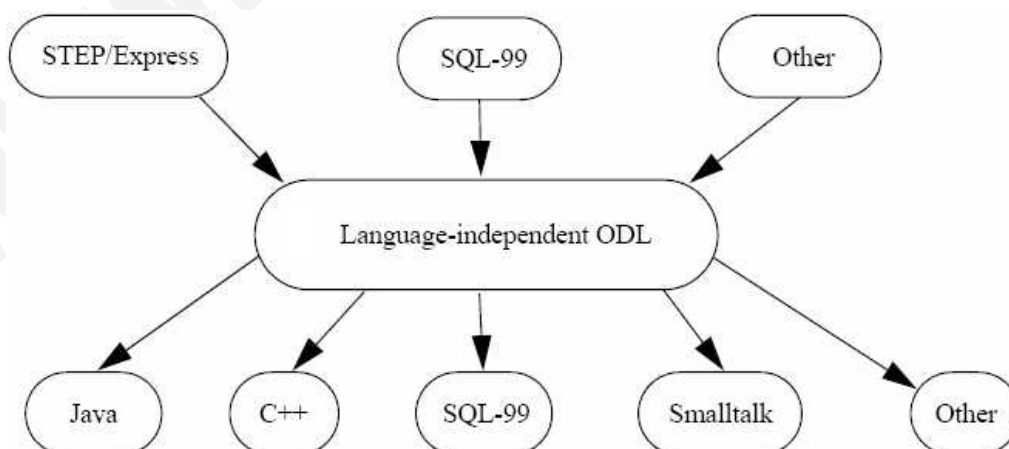
- Ref: Chapter 3 of R. Cattell, D. Barry, "The Object Data Standard: ODMG 3.0", Morgan Kaufman Publishers.
- PL independent specification to define schema, operations and state of a ODMS.
- Objective is to establish interoperability among systems.
- Two specification languages:
  - ✓ Object Definition Language (ODL)
  - ✓ Object Interchange Format (OIF)

### Object Definition Language (ODL)

#### ODL - Considerations

- ODL should be PL independent.
- ODL should not be a full PL.
- ODL should be compatible with the OMG's IDL.
- ODL should be extensible and practical.
- DBMSs provide DDLs; ODL serves the same purpose.
- Defines only signatures.
- No OML; includes specs of APIs for different PL bindings.

#### ODL Mapping to Other Languages



**Extended Backus Naur Form (EBNF)**

- Specification of ODL given in EBNF to define:
  - ✓ Interface
  - ✓ Class
  - ✓ Attributes
  - ✓ Operations
  - ✓ Exceptions

```

<interface> ::= <interface_dcl>
 | <interface_forward_dcl>
<interface_dcl> ::= <interface_header>
 { [<interface_body>] }
<interface_forward_dcl> ::= interface <identifier>
<interface_header> ::= interface <identifier>
 [<inheritance_spec>]
<class> ::= <class_dcl> | <class_forward_dcl>
<class_dcl> ::= <class_header> { <interface_body> }
<class_forward_dcl> ::= class <identifier>
<class_header> ::= class <identifier>
 [extends <scopedName>]
 [<inheritance_spec>]
 [<type_property_list>]

```

**Specification of ODL**

- Represented graphically, followed by schema definition.
- Types in rectangles and relationships using lines.

**Graphical Representation of Schema**

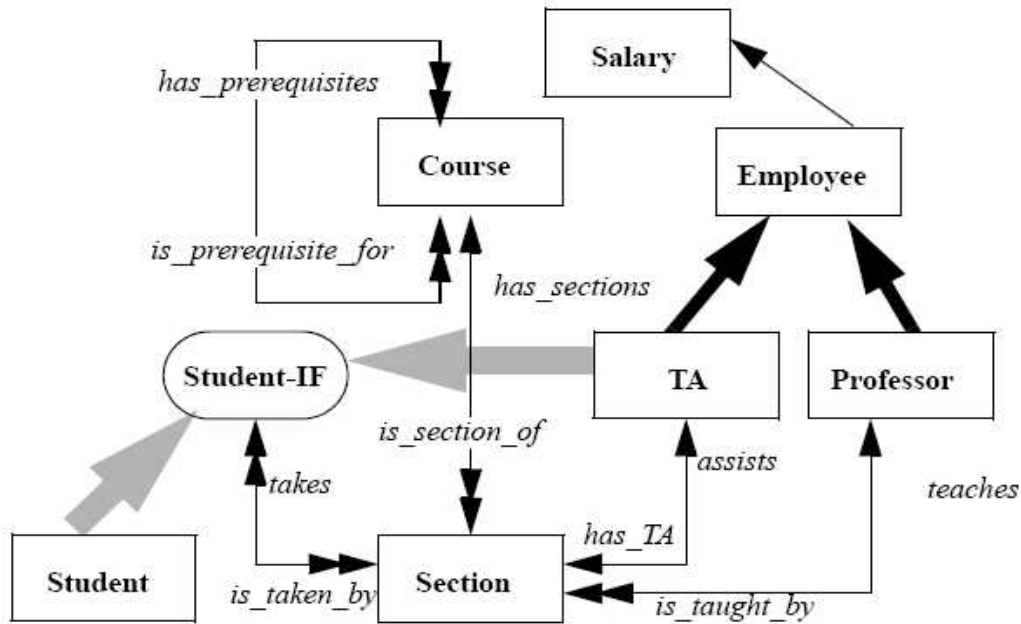
one-to-one 

one-to-many 

one-to-many 

is-a 

extends 



```

module ODMGExample {
 exception NoSuchEmployee();
 exception AlreadyOffered{};
 exception NotOffered{};
 exception IneligibleForTenure{};
 exception UnsatisfiedPrerequisites{};
 exception SectionFull{};
 exception CourseFull{};
 exception NotRegisteredInSection{};
 exception NotRegisteredForThatCourse{};

 struct Address {string college, string room_number; };

 class Department
 (extent departments)
 {
 attribute string name;
 relationship list<Professor> has_professors
 inverse Professor::works_in;
 relationship list<Course> offers_courses
 inverse Course::offered_by;
 };

```

```

class Course
(
 extent courses
)
{
 attribute string name;
 attribute string number;
 relationship Department offered_by
 inverse Department::offers_courses;
 relationship list<Section> has_sections
 inverse Section::is_section_of;
 relationship set<Course> has_prerequisites
 inverse Course::is_prerequisite_for;
 relationship set<Course> is_prerequisite_for
 inverse Course::has_prerequisites;
 boolean offer (in unsigned short semester)
 raises (AlreadyOffered);
 boolean drop (in unsigned short semester) raises (NotOffered);
};

```

- **classTA extends** Employee : StudentIF
 

```

{
 relationship Section assists
 inverse Section::has_TA;
 attribute string name;
 attribute string student_id;
 attribute struct Address dorm_address;
 relationship set<Section> takes
 inverse Section::is_taken_by;
};

```

### Object Interchange Format (OIF)

- A specification language to dump and load the current state of ODMS to or from a set of files.
- Should support all ODMS states.

### ODMS States

- States characterized using:
  - ✓ Object identifiers
  - ✓ Type bindings
  - ✓ Attribute values
  - ✓ Links to other objects

**Object Interchange Format**

- An OIF file contains object definitions specifying:
  - ✓ The type, attribute values, and relationships.
- Object identifiers are specified with object tag names unique to the OIF file(s).
- **An example definition:**
- Makha Person{};
- An uninitialized object.
- The object tag Makha is used to reference the defined object within the entire set of OIF files.

**Attribute Value Initialization**

- **The ODL definition**

```
interface Person {
 attribute string Name;
 attribute unsigned short Age;
};
```
- Object creation
- Salma Person{Name "Salma", Age 11}
- Salma Person{Age 11, Name "Salma"}
- Salma Person{"Salma", 11}
- Jamil(Saqib) Company{Saqib} // Copy initialization

**Lecture -24****Object Interchange Format (OIF)**

- Attribute initialization.
- Names may be given or ignored if order (in definition) followed.
  - ✓ Boolean literals
  - ✓ Character literals
  - ✓ Integer literals (decimal, octal, hexadecimal), Float literals
  - ✓ String literals

**Initializing Structured Type**

- struct PhoneNumber {
   
        unsigned short CountryCode;
   
        unsigned short AreaCode;
   
        unsigned short PersonCode;
   
};
- struct Address {
   
        string Street;
   
        string City;
   
        PhoneNumber Phone;
   
};
- interface Person
   
        attribute string Name;
   
        attribute Address PersonAddress;
   
};
- Sara Person {Name "Sara",
   
        PersonAddress {Street "Hospital Road",
   
                        City "Rahim Yar Khan",
   
                        Phone {CountryCode 92, AreaCode 0122,
   
                                PersonCode 1234}
   
        }
   
}

**Array Attributes**

- interface Engineer {
   
        attribute unsigned short PersonID[3];
   
};
   
Nadeem Engineer{PersonID{[0] 450, [2] 270}}
- Any attributes not specified remain uninitialized.
- interface Professor: Person {
   
        attribute set<string> Degrees;
   
};
   
AbdulQadir Professor{Degrees {"Masters", "PhD"}}

- struct Point {  
    float X;  
    float Y;  
};
- interface Polygon {  
    attribute array<Point> RefPoints; };
- P1 Polygon {RefPoints{[5]{X 7.5, Y 12.0}, [11]{X 22.5, Y 23.0}}

#### Link Definitions

- interface Person{  
    relationship Company Employer  
    inverse Company::Employees;  
};  
Nadeem Person{Employer Muneer}
- Link established between the object tagged Nadeem and the object tagged Muneer.
- interface Company {  
    relationship set<Person> Employees  
    inverse Person::Employer;  
};  
Nadeem Company{Employees {Saleem, Liaqat, Amjad}}

#### Command Line Utilities

- odmsdump <odms\_name>
- Object tag names are created automatically.
- odmsload <odms\_name> <file 1> ... <file n>
- populates the ODMS with the objects defined in the specified files.

#### Object Query Language (OQL)

- Relies on the ODMG Object Model.
- Very close to SQL-92.
- Provides high-level primitives to deal with sets of objects and also structures, lists, and arrays and very efficiently.
- Can be invoked from PL.
- Does not provide update ops.
- A functional language where operators can freely be composed as long as the operands respect the type system.
- OQL is computationally complete. It is a simple-to-use query language.
- Can be used as standalone or as embedded in a PL.

**Query Input and Result**

- `select distinct x.age from Persons x  
where x.name = "Saleem"`  
Returns set of integers.
- `select distinct struct(a: x.age, s: x.sex) from Persons x  
where x.name = "Saleem"`  
Returns set of structures.
- Select can be used in From part  
`select struct (a: x.age, s: x.sex)  
from (select y from Employees y where y.seniority = "10") as x where x.name = "Saleem"`
- Short-cut notation  
`Chairman; // retrieves chairman object`  
`Chairman.subordinates; // set of subordinates from chairman object`

**Creating Objects**

- `Person(name: "Saleem", birthdate: date '19646-3-28', salary: 100,000) // object with id`
- `struct (a: 10, b: "Pat") // structure with two fields, no id`
- Query may result (atomic or collection) object with or without OID.

**Path Expressions**

- OM supports "." notation  
`p.spouse.address.city.name // a 1-1 link`
- For one to many we have to adopt Select statement  
`select c.name from p.children c // return bag`  
`select distinct c.name from p.children c // set`  
`select c.address from Persons p, p.children c // traverses two collections`

**Predicates**

- `select c.address from Persons p, p.children c where p.address.street = "Main Street" and  
count(p.children) >= 2 and c.address.city != p.address.city`
- Boolean operators
  - ✓ X andthen Y
  - ✓ X or else Y
- `select e from Employees e where e.address != Null andthen e.address.city="Multan"`

**Undefined Values**

- Result of accessing property of a nil object.
  - ✓ **UNDEFINED**
  - ✓ `is_undefined()`

**Method Call**

- Can be called.
- Notation as attrib/relationship  
select max(select c.age from p.children c) from Persons p where p.name = "Paul"
- Conflict resolution through ()

**Polymorphism**

- Person object can call methods of Person, Employee or Student  
select p.activities from Person p
- Late Binding

**Object Query Language**

- This follows the formal definitions of different components of OQL and the syntax in BNF, that we can skip.

**Lecture -25****C++ Binding**

- Reference: Chapter 5 of ODMG 3.0.
- Describes C++ binding for ODL/OML.

**C++ Binding**

- Basic Principal
  - ✓ Same syntax for ODL/OML
  - ✓ Same set of data types
- Persistence-capable Classes
- Smart Pointer or Ref
  - ✓ `d_Ref<Professor> profP;`
  - ✓ `profP->get_tenure();`

**Mapping of Object Model into C++**

- Components of ODMG Object model map into C++ model.
- Objects and literals, Structure, Implementation, Collection, Array, Relationships, Prefix, namespace etc.

**C++ ODL**

- Provides a description of DB schema as a set of object classes.
- `class Professor : public d_Object { public:`

```

// properties:
d_UShort age;
d_UShort id_number;
d_String office_number;
d_String name;
d_Rel_Ref<Department, _professors> dept;
d_Rel_Set<Student, _advisor> advisees;
// operations:
void grant_tenure();
void assign_course(Course &);
private:
...
};
```
- Fixed Length Types
  - ✓ `d_Short`, `d_Long`, `d_Ushort`, `d_Ulong`, `d_Float`, `d_Double`, `d_Char`, `d_Octet`, `d_Boolean`, `d_True` or `d_False`
- Structured Literal Types
  - ✓ `d_String`, `d_Interval`, `d_Date`, `d_Time`, `d_Timestamp`.

- ```
class d_String {
    public: // constructor and destructor
        d_String();
        d_String(const d_String &);
        d_String(const char *);
        ~d_String();

    // operations
        d_String & operator=(const d_String &);
        d_String & operator=(const char *);
        operator const char *() const;
        char & operator[ ](unsigned long index);
        unsigned long length() const;
    friend d_Boolean operator==(const d_String &sL, const d_String &sR);
    friend d_Boolean operator==(const d_String &sL, const char *pR);
}
```

Relationships

- Like attribute declaration but also specifies traversal path
 - ✓ `d_Rel_Ref<T, const char *>`
 - ✓ `d_Rel_Set<T, const char *>`
 - ✓ `d_Rel_List<T, const char *>`

```
extern const char _dept [ ], _professors [ ];
extern const char _advisor [ ], _advisees [ ];
extern const char _classes [ ], _enrolled [ ];
```
- ```
class Department : public d_Object {
 public:
 d_Rel_Set<Professor, _dept> professors; };
class Professor : public d_Object {
 public:
 d_Rel_Ref<Department, _professors> dept;
 d_Rel_Set<Student, _advisor> advisees; };
const char _dept [] = "dept";
const char _professors [] = "professors";
```

### C++ OML

- A guiding principle: syntax to create, delete, identify, reference, get/set property values, and invoke operations on a persistent object should be same.
- A single expression may freely intermix references to persistent and transient objects.

**Object Operations**

- Created using overloaded “new” operator.
  1. `void * operator new(size_t size);`
  2. `void * operator new(size_t size, const d_Ref_Any &clustering, const char* typename);`
  3. `void * operator new(size_t size, d_Database *database, const char* typename);`
- `d_Database *yourDB, *myDB; // assume these get initialized properly`
  1. `d_Ref<Schedule> temp_sched1 = new Schedule;`
  2. `d_Ref<Professor> prof2 = new(yourDB, "Professor") Professor;`
  3. `d_Ref<Student> student1 = new(myDB, "Student") Student;`
  4. `d_Ref<Student> student2 = new(student1, "Student") Student;`
- Objects, once created, can be deleted in C++ OML using the `d_Ref::delete_object` member function.
- Persistent or transient.
 

```
d_Ref<Professor> obj_ref;
obj_ref.delete_object();
```

**Object Modification**

- Modified within methods.
- Changes permanent when transaction is committed.
- ODMS need to be intimated through.
 

```
obj_ref->mark_modified();
```
- May be omitted through a preprocessor switch.

## Lecture -26

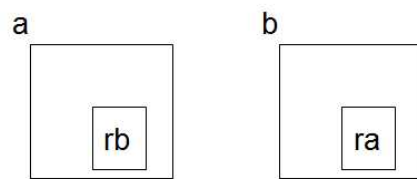
## Manipulating Properties

## Attributes

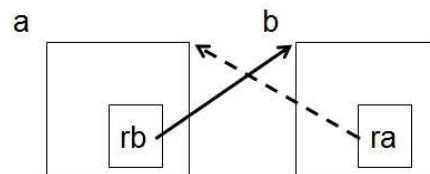
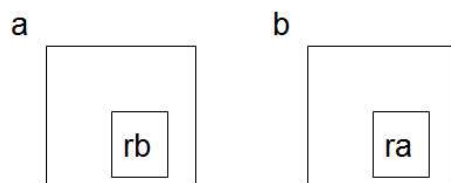
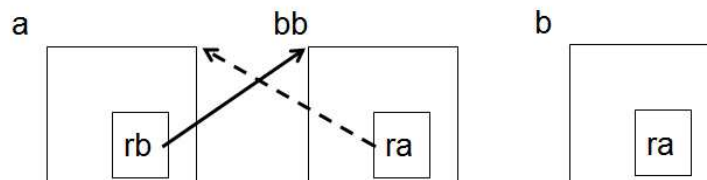
- Attributes: standard C++ notation.
  - ✓ `prof->id_number = next_id;`
  - ✓ `cout << prof->id_number;`
- Before modifying must call `mark_modified` method.
- Persistence-enabled can contain both types of objects.

## Relationships

- Relationships: ODL specifies and OML manipulates.
- All four types supported.
- `extern const char _ra [], _rb [] ;`
- class A {
  - `d_Rel_Ref<B, _ra> rb;`
- };
- class B {
  - `d_Rel_Ref<A, _rb> ra;`
- };
- `const char _ra [] = "ra";`
- `const char _rb [] = "rb";`
- `d_Ref<A> a = new A;`
- `d_Ref<A> aa = new A;`
- `d_Ref<B> b = new B;`
- `d_Ref<B> bb = new B;`



1:1 N

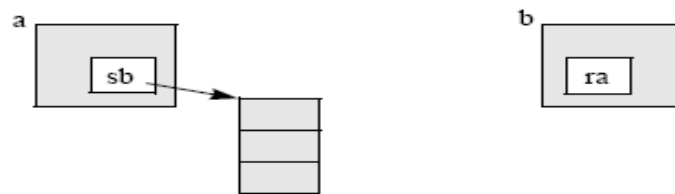
`a.rb = &b;``a.rb.clear();``a.rb = &bb;`

- All relationships deleted with the deletion of object.

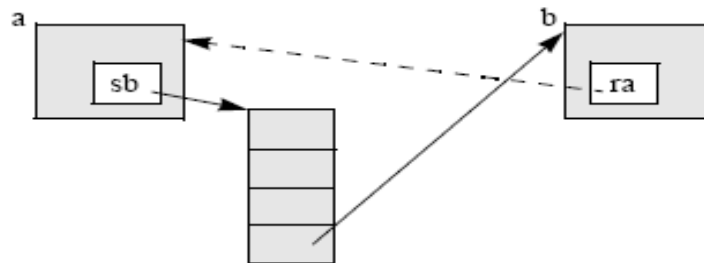
**Relationships**

- 1:m Relationship  

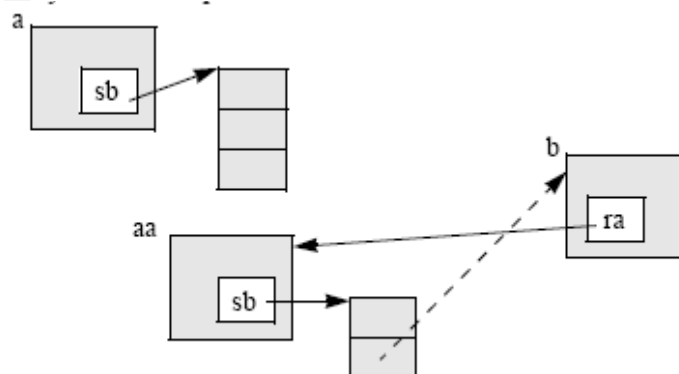
```
extern const char _ra [], _sb [] ;
class A {
 d_Rel_Set<B, _ra> sb;
};
class B {
 d_Rel_Ref<A, _sb> ra;
};
const char _ra[] = "ra";
const char _sb[] = "sb";
```



- `a.sb.insert_element(&b);`



- `a.sb.remove_element(&b);` or `b.ra.clear();`
- Same effect with `b.ra = &a;`
- `b.ra = &aa`



**SET**

class List: Collection {

```

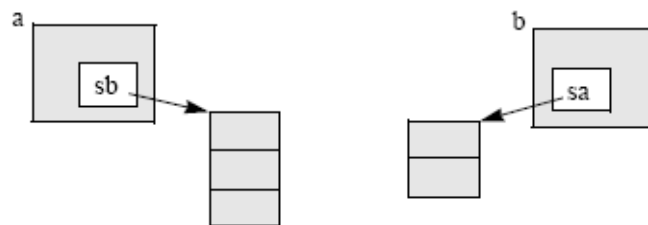
 exception InvalidIndex {undigned long index; };
 attribute list<t> value;
 void remove_element_at(in unsigned long index) raises(InvalidIndex);
 Object retrieve_element_at(in unsigned long index) raises(InvalidIndex);
 void replace_element_at(in Object element, in unsigned long index)
 raises(InvalidIndex);

 void insert_element_after(in Object element, in unsigned long index) raises(InvalidIndex);
 void insert_element_before(in Object element, in unsigned long index) raises(InvalidIndex);
 void insert_element_first(in Object element)
 void insert_element_last(in Object element)
 }
```

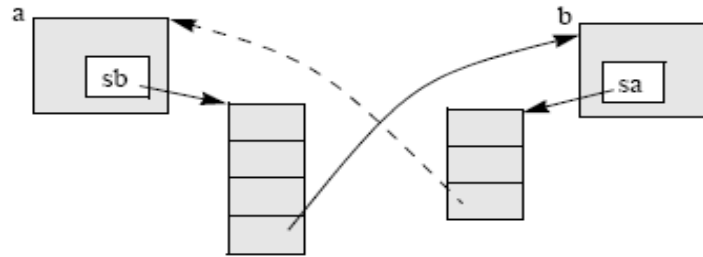
- extern const char \_ra [ ], \_listB [ ];
- class A {
 d\_Rel\_List<B, \_ra> listB;
 };
 class B {
 d\_Rel\_Ref<A, \_listB> ra;
 };
 const char \_ra [ ] = "ra";
 const char \_listB [ ] = "listB";
 d\_Ref<A> a = new A;
 d\_Ref<B> b = new B;

**Methods**

- a.listB.insert\_element\_first(&b);
- a.listB.insert\_element\_after(&b, 3);
- a.listB.retrieve\_element\_at(k);
- Initially

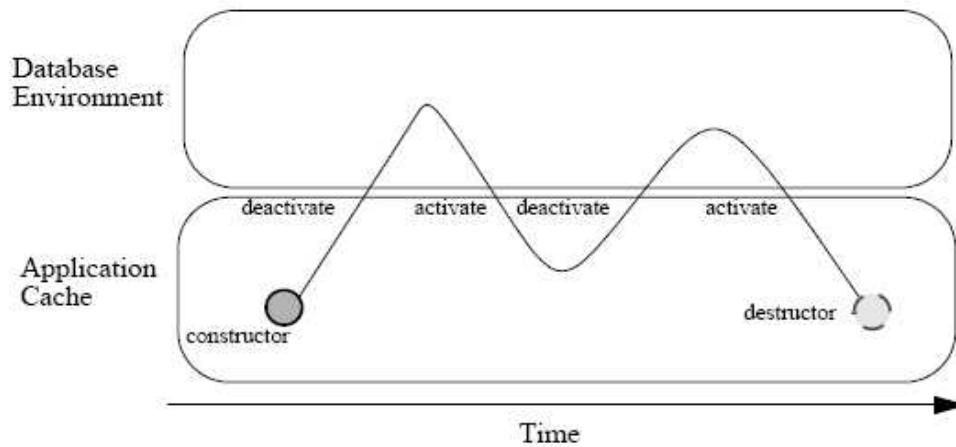


- `a.sb.insert_element_last(&b)`



### Operations

- Operations: All that we normally perform on transient objects in C++.
- `d_object` class to all both `t` and `p` objects.
- ODMG uses (de)activate when `T` object moves in & out of Application cache.
- Use of constructor and destructor.



### Reference Classes

- Created to refer to persistent objects.
- Work same as pointers with one difference; support of referential integrity.
- Definition of collection classes also in OML; same as in specs.

### Shallow & Deep Copy

- Applies to all collection classes:
  - ✓ Structure, Array, Set, List, Bag, Dictionary etc.
- Different declarations.
  - ✓ `d_Ref <Professor> faculty;`
  - ✓ `d_Set<d_Ref<Professor> > faculty1;`
  - ✓ `d_Set<Professor> faculty2;`

**C++ OML**

- Similarly d\_Database type has been defined for DB operations.
- DB cannot be created from C++, it has to be created using ODMS.
- In program, it has to be opened and closed.
- class d\_Database {
  - public:
 

```
static d_Database * const transient_memory;
enum access_status { not_open, read_write, read_only, exclusive };
d_Database();
void open(const char * database_name,
access_status status = read_write);
void close();
void set_object_name(const d_Ref_Any &theObject,
const char* theName);
void rename_object(const char * oldName, const char * newName);
d_Ref_Any lookup_object(const char * name) const;
```
  - private:
 

```
d_Database(const d_Database &);
d_Database & operator=(const d_Database &);
```

**C++ OQL**

- Query method has been defined in d\_Collection class.
 

```
int query(d_Collection<T> &result, const char* predicate) const;
d_Bag<d_Ref<Student> > mathematicians;
Students->query(mathematicians, "exists s in this.takes: s.section_of.name = \"math\"");
```
- d\_oql\_execute function.
- Query has to be constructed and then executed.
- class d\_OQL\_Query {
  - public:
 

```
d_OQL_Query();
d_OQL_Query(const char *s);
d_OQL_Query(const d_String &s);
d_OQL_Query(const d_OQL_Query &q);
~d_OQL_Query();
d_OQL_Query & operator=(const d_OQL_Query &q);
void clear();
friend d_OQL_Query & operator<<(d_OQL_Query &q, const char *s);
friend d_OQL_Query & operator<<(d_OQL_Query &q, const d_String &s);
friend d_OQL_Query & operator<<(d_OQL_Query &q, d_Char c);
.....
```

**Lecture -27****OO support in Oracle**

- Ref: W. Rahayu, D. Taniar, E. Pardede, (editors) “Object-oriented Oracle”, IRM Press, 2006.

**Object Types and UDTs**

- “Create Type” is used to create a type, that can then be used as a generic type to create a table.
- “As Object” creates type as object.
- General Syntax:  
CREATE [OR REPLACE] TYPE <ob\_Name> AS OBJECT (attribute attribute type, ..., attribute attribute type)

- Example:  
CREATE OR REPLACE TYPE PersonType AS OBJECT  
    (pid VARCHAR2(10),  
    pname VARCHAR2(30))  
CREATE TABLE Course  
    (cld VARCHAR2(10),  
    cname VARCHAR2(20),  
    instructor Person\_T);

**Collection Types**

- We can create a type as “varying array” varray.
- General Syntax:  
CREATE [OR REPLACE] TYPE <obj\_Name> AS VARRAY(n) OF (object/data type)
- Example:  
CREATE OR REPLACE TYPE Ustaads AS VARRAY(4) OF PersonType  
CREATE TABLE Course  
    (cld VARCHAR2(10),  
    cname VARCHAR2(20),  
    lecturer Ustaads);

**Nested Tables**

- General Syntax:  
CREATE [OR REPLACE] TYPE <ob\_table\_Name> AS TABLE OF (ob\_Name)  
CREATE TABLE <tab\_Name>  
    (attribute attribute type, ...,  
    outer nested item object table schema);  
NESTED TABLE <outer nested item>  
    STORE AS <outer storage table schema>  
(NESTED TABLE <inner nested item>  
    STORE AS <inner storage table schema>);

- Example:  
CREATE OR REPLACE TYPE PersonType AS OBJECT (pId VARCHAR2(10),  
pName VARCHAR2(30))
- CREATE OR REPLACE TYPE PersonTableType AS TABLE OF PersonType
- CREATE TABLE Course  
(cId VARCHAR2(10),  
cName VARCHAR2(20),  
lecturer PersonTableType)
- NESTED TABLE lecturer STORE AS PersonTab;

### Object Identifier

- Not supported as such.
- Primary Key concept can be utilized to serve the purpose.
- Specially to implement inheritance.
- Example:  
CREATE OR REPLACE TYPE PersonType AS OBJECT (pId VARCHAR2(10),  
pName VARCHAR2(30))  
  
CREATE OR REPLACE TYPE EmpType AS OBJECT (pId VARCHAR2(10),  
title VARCHAR2(10),  
Sal NUMBER)  
  
CREATE TABLE Person OF PersonType  
(pId NOT NULL, PRIMARY KEY (pId));  
  
CREATE TABLE Emp OF EmployeeType  
(pId NOT NULL, PRIMARY KEY (pId),  
  
FOREIGN KEY (pId) REFERENCES Person(pId));

### Relationships using Ref

- Ref keyword used to reference from one object to other.
- Can be used in place of Join.
- General Syntax:  
CREATE TABLE <table schema> (object REF (ob\_Name) [SCOPE IS (tab\_Name)]);
- Example:  
CREATE OR REPLACE TYPE PersonType AS OBJECT (pId VARCHAR2(10), pname VARCHAR2(30))
- CREATE TABLE AcadStaff OF PersonType;

- CREATE TABLE Course  
 (cId VARCHAR2(10),  
 cName VARCHAR2(20),  
 lecturer REF PersonType SCOPE IS AcadStaff);
- “SCOPE IS” is optional.
- SELECT C.cName FROM Course C WHERE C.lecturer.pName = 'Jalil';

### Inheritance

- A more genuine approach.
- Superclass definition contains “Not Final”; Final by default.
- Subclass definition contains “Under” keyword.
- General Syntax:  
 CREATE [OR REPLACE] TYPE <super-type ob\_Name> AS OBJECT  
 (key attribute attribute type,  
 attribute attribute type,...,  
 attribute attribute type) [FINAL| NOT FINAL]  
  
 CREATE [OR REPLACE] TYPE <sub-type ob\_Name> UNDER <super-type Ob\_Name>  
 (additional attribute attribute type, ...,  
 additional attribute attribute type)  
 [FINAL| NOT FINAL]  
  
 CREATE TABLE <super-type tableName> OF <super-type ob\_Name>  
 (key attribute NOT NULL,  
 PRIMARY KEY (key attribute));
- Example:  
 CREATE OR REPLACE TYPE PersonType AS OBJECT  
 (pId VARCHAR2(10),  
 pName VARCHAR2(20),  
 address VARCHAR2(35)) NOT FINAL
- CREATE OR REPLACE TYPE StudentType UNDER PersonType  
 (course VARCHAR2(10),  
 year NUMBER)
- CREATE TABLE Person OF PersonType  
 (pId NOT NULL,  
 PRIMARY KEY (pId));

**Encapsulation**

- Two approaches:
  - ✓ Stored Procedures
  - ✓ Member Procedures

**Stored Procedures**

- Use of “Create Procedure”.
- Parameters support
  - ✓ In: can take the value inside P/F only; is read only, can’t be changed.
  - ✓ Out: brings the value out, is write only.
  - ✓ InOut; mixture of both.
- General Syntax:
 

```
CREATE [OR REPLACE] PROCEDURE <pr_Name> [parameter [{IN | OUT | IN OUT}] parameter
type,
...,
parameter [{IN | OUT | IN OUT}] parameter type]) AS [local variables]
BEGIN
 <procedure body>;
END <pr_Name>;
GRANT EXECUTE ON <pr_Name> TO <user>;
```
- Example:
 

```
CREATE OR REPLACE PROCEDURE UpdateEmp(updId Emp.eId%TYPE, salVal Emp.sal%TYPE) AS
BEGIN
 UPDATE Emp Set Sal = salVal
 WHERE eId = updId;
END UpdateEmp;
/
```
- GRANT EXECUTE ON UpdateEmp TO EVP;

**Stored Function**

- Similar to stored procedure, except that function returns a value.
- So, we need to specify the return type.

**Lecture -28****OO support in Oracle**

- Ref: W. Rahayu, D. Taniar, E. Pardede, (editors) “Object-oriented Oracle”, IRM Press, 2006.

**Member Procedure or Function**

- **General Syntax**
- CREATE [OR REPLACE] TYPE <object schema> AS OBJECT  
 (attribute attribute types,.....attribute attribute types,  
 MEMBER PROCEDURE <procedure name>  
 [(parameter [{IN | OUT | IN OUT}] parameter  
 type,.....parameter [{IN | OUT | IN OUT}] parameter type)],  
 MEMBER FUNCTION <function name>  
 [(parameter [{IN}] parameter type,.....parameter [{IN}] parameter type)]  
 RETURN datatype);  
 /
- **Implementation**
- CREATE [OR REPLACE] TYPE BODY (object schema) AS  
 MEMBER PROCEDURE <member procedure name>  
 [parameter [{IN | OUT | IN OUT}] parameter type,.....parameter [{IN | OUT | IN OUT}]  
 parameter type)] IS  
 [local variables]  
**BEGIN**  
     <procedure body>;  
 END  
     <member procedure name>;
- MEMBER FUNCTION <function name>  
 [parameter [{IN}] parameter type,.....parameter [{IN}] parameter type)]  
 RETURN datatype IS  
 [local variables]  
**BEGIN**  
     <procedure body>;  
 END  
     <member function name>;  
**END;**
- /

**Visibility**

- Public
- Private
- Protected

**Overriding**

- 
- 

**OO Features in RDBMS**

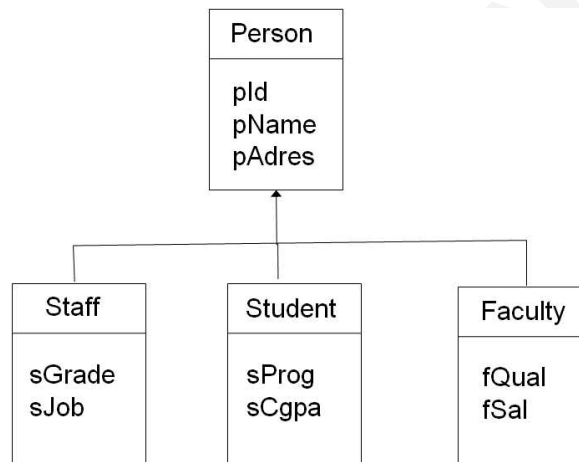
- 
- 

**Inheritance**

- Union Inheritance
- Mutual-Exclusion Inheritance
- Partition Inheritance

**Union Inheritance**

- An object can belong to more than one subclass or none

**Inheritance**

- Use of “Under”.
- Superclass definition contains “Not Final”; Final by default.
- Subclass definition contains “Under” keyword.
- Example:

```

CREATE OR REPLACE TYPE Person_T AS OBJECT(id VARCHAR2(10),
name VARCHAR2(20),
address VARCHAR2(35)) NOT FINAL
/

```

```

CREATE OR REPLACE TYPE Student_T UNDER Person_T(course VARCHAR2(10),
year VARCHAR2(4))
/

```

- CREATE TABLE Person OF Person\_T (id NOT NULL, PRIMARY KEY (id));
- CREATE TABLE <super-type tabName> OF <super-type ob\_Name>  
(key attribute NOT NULL,  
PRIMARY KEY (key attribute));
- Example:  
CREATE OR REPLACE TYPE PersonType AS OBJECT  
(pId VARCHAR2(10),  
pName VARCHAR2(20),  
address VARCHAR2(35)) NOT FINAL  
/

**Lecture -29****Mutual-Exclusion Inheritance**

- 
- 
- 

**Mutual-Exclusion Inheritance using OO Features**

- 
- 
- 

**Implementing Mutual Exclusion Inheritance using OO Features**

- Using under clause.
- Introducing a type attribute.
- Table only for the super type not for the subtypes.
- //Create Person Type and Table  

```
CREATE OR REPLACE TYPE Person_TM AS OBJECT
 (pId VARCHAR2(10), pName VARCHAR2(20),
 pAdres VARCHAR2(35))

CREATE TABLE PersonM OF Person_TM
 (pId NOT NULL, perType VARCHAR2(10)
 CHECK(perType IN ('Student', 'Staff', 'Faculty', NULL), PRIMARY KEY (pId));
// Create student, staff and faculty types;
no tables

CREATE OR REPLACE TYPE Student_TM UNDER Person_TM
 (sProg VARCHAR2(sCgpa NUMBER)

CREATE OR REPLACE TYPE Faculty_TM UNDER Person_TM
 (fQual VARCHAR2(10), fSal NUMBER)

CREATE OR REPLACE TYPE Staff_TM UNDER Person_TM
 (sGrade VARCHAR2(10), sJob VARCHAR2(20))
```

**Managing Additional Attributes of Sub-Class**

- 
- 
-

**Partition Inheritance**

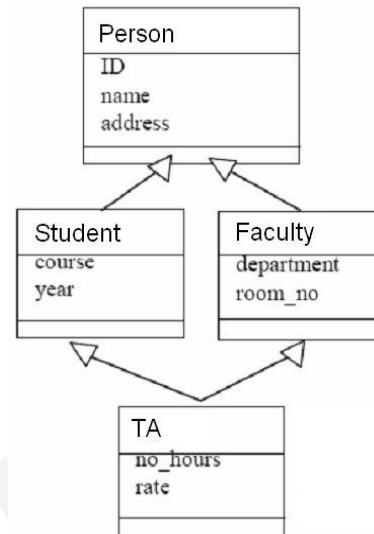
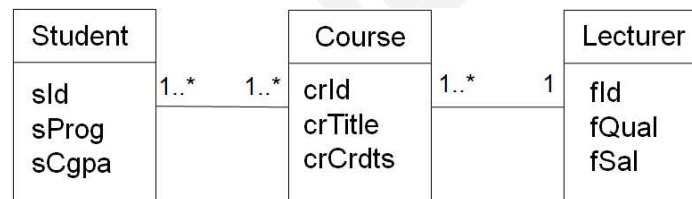
- An object can only be of one and only one type of subclasses.
- No Superclass type.
- No more than one subclass type.
- Implementation same as Mutual Exclusion.
- Type attribute cannot have Null value; only subclass types.
- In OO implementation, table against only super class.

**Inheritance**

- Union Inheritance
- Mutual-Exclusion Inheritance
- Partition Inheritance

**Multiple Inheritance**

- Using same PK/FK method in Relational approach.

**Association**

- All class should be visible all time. You can make objects close to each other and make the lines between Student and Course from top, and from Course to Lecturer from bottom to achieve that Simple animation just connect the lines.
- The text is:
  - 1..\*
  - 1
- Student
 

|       |
|-------|
| sId   |
| sProf |
| sGpa  |
- Course
 

|         |
|---------|
| crId    |
| crTitle |
| crCrdts |
- Lecturer
 

|       |
|-------|
| fld   |
| fQual |
| fSal  |

**Association**

- Relational approach uses PK/FK. combination for the implementation.

**OO Features in Oracle**

- Ref stores reference or OID of the other object.
- Ref can store only a single reference.
- Value is system generated.

```
CREATE OR REPLACE TYPE Student_T AS OBJECT
 (sId VARCHAR2(10), sProg VARCHAR2(30), sCgpa NUMBER)
```

```
CREATE TABLE Student OF Student_T;
```

```
CREATE OR REPLACE TYPE Faculty_T AS OBJECT
 (fId VARCHAR2(10), fQual VARCHAR2(10), fSal NUMBER)
```

```
CREATE TABLE Faculty OF FACULTY_T
 // Course_T contains implementation of 1-n relationship
```

```
CREATE OR REPLACE TYPE Course_T AS OBJECT
 (crId VARCHAR2(10), crTitle VARCHAR2(20), crCrdts NUMBER,
 lecturer REF Faculty_T SCOPE IS Faculty);
```

- CREATE TABLE Course OF Course\_T  
// ENROLL implements Many to Many relationship
- CREATE TABLE Enroll (stId REF Student\_T, crId REF Course\_T)

**Assigning Value and Accessing Value of REF**

- **Assigning Value of REF**  
insert into course values ('C101', 'Database', 3, (select ref(a) from Faculty a where a.fId = 'F101'))  
insert into course select 'C105', 'OS', 3, ref(a) from Faculty a where fId = 'P102'  
select c.crId, c.crTitle, c.lecturer.fQual  
from course c

**Containment Relationships (Aggregation & Composition)****Aggregation**

- Exclusive
- Non-Exclusive

**Non-Exclusive Aggregation**

- Existence dependent (Composition), non exclusive aggregation.

**Cluster**

- 
-

**Lecture -30****Containment Relationships****(Aggregation & Composition)**

- 
- 

**Non-Exclusive Aggregation**

- Existence dependent (Composition), non exclusive aggregation

**Existence Dependant Aggregation**

- Exclusive
- Non-Exclusive

**Cluster**

- CREATE CLUSTER Dep\_Cluster (dep\_id VARCHAR2(3));
- CREATE TABLE Dept  
(dep\_id VARCHAR2(3) NOT NULL,  
location VARCHAR2(20),  
PRIMARY KEY (dep\_id))  
CLUSTER Dep\_Cluster(dep\_id);
- CREATE TABLE Emp  
(dep\_id VARCHAR2(3) NOT NULL,  
emp\_id VARCHAR2(3) NOT NULL,  
eName VARCHAR2(30),  
eAddress VARCHAR2(20),  
PRIMARY KEY (dep\_id, emp\_id),  
FOREIGN KEY (dep\_id)  
REFERENCES Dept (dep\_id) on delete cascade)  
CLUSTER Dep\_Cluster(dep\_id);
- CREATE INDEX Dep\_Cluster\_Index ON  
CLUSTER Dep\_Cluster;

**Non-Exclusive Aggregation**

- Existence dependent (Composition), non exclusive aggregation

**Exclusive Aggregation**

- Existence dependent (Composition), exclusive aggregation.  
CREATE CLUSTER Dep\_Cluster1 (dep\_id VARCHAR2(3));

```
CREATE TABLE Dept11 (dep_id VARCHAR2(3) NOT NULL, location VARCHAR2(20),
PRIMARY KEY (dep_id))
CLUSTER Dep_Cluster1 (dep_id);
```

```
CREATE TABLE Emp31 (dep_id VARCHAR2(3) NOT NULL, emp_id VARCHAR2(3) NOT NULL,
eName VARCHAR2(30), eAddress VARCHAR2(20), PRIMARY KEY (emp_id),
FOREIGN KEY (dep_id) REFERENCES Dept11 (dep_id) on delete cascade)
```

```
CREATE INDEX Dep_Cluster_Index1 ON CLUSTER Dep_Cluster1;
```

### Nested Tables

- 

### Using Nested Tables

```
CREATE OR REPLACE TYPE Person_tc as object
(person_id varchar2(10), person_name varchar2(20))
```

```
CREATE OR REPLACE TYPE Person4c AS TABLE OF Person_Tc
```

```
CREATE TABLE Cors3(course_id
VARCHAR2(10), course_name
VARCHAR2(20), lecturer Person4c) NESTED
TABLE lecturer STORE AS Person_tab4
```

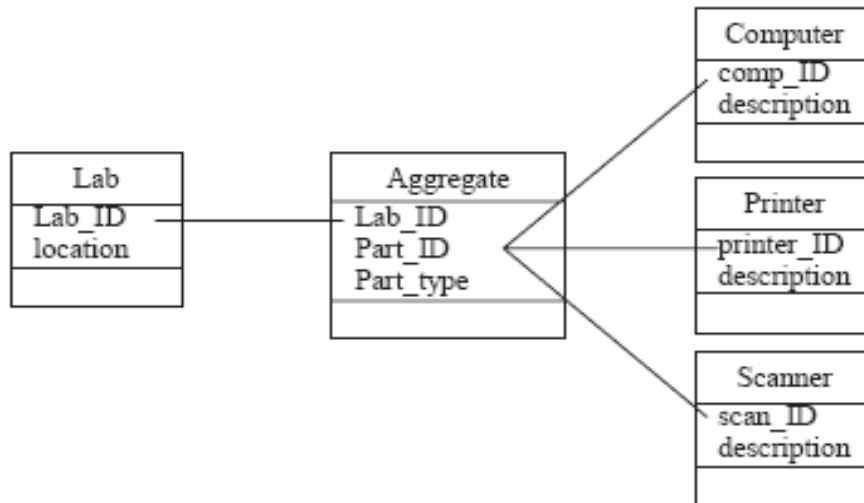
```
INSERT INTO cors3 VALUES ('C101', 'Database',
person4c(person_t('P101', 'Nayer'),
person_t('p102', 'Masood'), person_t('p103', 'Dar')))
```

### Non-Existence Dependent Aggregation

- Using the PK/FK combination to form an aggregate object.

### Computer Lab



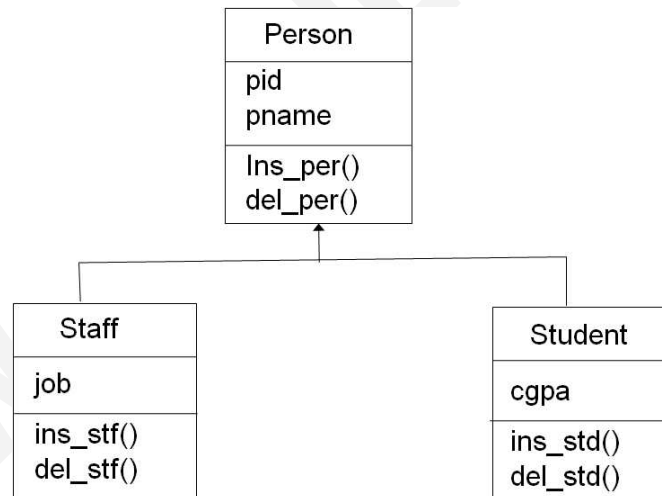


### Object-Oriented Methods

- Chapter 4 is to explain the difference between the two, i.e., stored procedures and member procedures.

### Generic Methods

- Written to access the attributes of tables; they are member procedures/functions.
- To implement encapsulation.



### Implementing Methods

- Union Inheritance.
- Relational Approach.
- Need to define type; because we are interested in member methods.
- // Define type Person and table;  
Create or replace type Person\_tu as object (pid varchar2(10), pname varchar2(20) ,  
MEMBER PROCEDURE Insert\_Person( new\_id IN VARCHAR2, new\_name IN VARCHAR2),  
MEMBER PROCEDURE Delete\_Person);

- ```
CREATE TABLE PersonU OF Person_TU (pid NOT NULL, PRIMARY KEY (pid));
```
- ```
// Create type body for Person
CREATE OR REPLACE TYPE BODY Person_TU AS
MEMBER PROCEDURE Insert_Person(new_id IN VARCHAR2, new_name IN VARCHAR2) IS
 BEGIN
 INSERT INTO PersonU
 VALUES (new_id, new_name);
 END Insert_Person;
MEMBER PROCEDURE Delete_Person IS
 BEGIN
 DELETE FROM PersonU
 WHERE PersonU.pid = self.pid;
 END Delete_Person;
END;
```
  - ```
// Create type student and table
CREATE OR REPLACE TYPE Student_TU AS OBJECT
    (sid VARCHAR2(10), cgpa number(5,2),
MEMBER PROCEDURE Insert_StudentU(
    new_id IN VARCHAR2, new_name IN VARCHAR2,
    cgpa IN number),
MEMBER PROCEDURE Delete_Student);
CREATE TABLE StudentU OF Student_TU
    (sid NOT NULL, PRIMARY KEY (sid),
    FOREIGN KEY (sid) REFERENCES PersonU(pid)
    ON DELETE CASCADE);
```
 - ```
// Create body for insert_studentu
```
  - ```
CREATE OR REPLACE TYPE BODY Student_TU AS
MEMBER PROCEDURE Insert_StudentU(new_id IN VARCHAR2, new_name IN VARCHAR2,
                                cgpa number) IS loc_var number;
    BEGIN
        select count(*) into loc_var from personU where pid = new_id;
        If loc_var = 0 then
            INSERT INTO PersonU VALUES (new_id, new_name);
        end if;
        INSERT INTO StudentU VALUES (new_id, cgpa);
    END Insert_StudentU;
```

Lecture -31**Types**

- Created Types/Member Procedures:
 - ✓ Person
 - ✓ Student

Controlling Multiple Inheritance

- // create body for insert_studentu
 CREATE OR REPLACE TYPE BODY Student_TU AS
 MEMBER PROCEDURE Insert_StudentU(new_id IN VARCHAR2,
 new_name IN VARCHAR2,cgpa number) IS
 loc_var number;
 BEGIN
 select count(*) into loc_var from personU where pid = new_id;
 If loc_var = 0 then
 INSERT INTO PersonU VALUES (new_id, new_name);
 end if;
 INSERT INTO StudentU VALUES (new_id, cgpa);
 END Insert_StudentU;
- // body for delete student
 MEMBER PROCEDURE Delete_Student IS
 BEGIN
 DELETE FROM StudentU
 WHERE StudentU.sid = self.sid;
 DELETE FROM PersonU WHERE
 (PersonU.pid = self.sid) AND (PersonU.pid NOT IN
 (SELECT StaffU.sid FROM StaffU WHERE StaffU.sid self.sid));
 END Delete_Student;
 END; // for body

Encapsulation

- Calling insert methods**
- declare
 pr person_tu := person_tu(NULL, NULL);
 std student_tu:= student_tu (NULL, NULL, NULL);
 stf staff_tu:=staff_tu(NULL, NULL, NULL);
 Begin
 pr.insert_person('P11', 'Amjad');
 std.insert_studentU('P11', 'Amjad', 3.2);
 stf.insert_staffU('P11', 'Amjad', 'Manager');
 end;

- **Calling delete methods**

declare

```
std student_tu:= select * from studentU where stdentU.sid = 'P11';
begin
    std.delete_studentU();
end;
```

Union Inheritance with OO Features

- Tables for all three types will be created
- Under clause will include all the members of superclass into subclasses
- Same happens with the tables
- So all three tables created
- Duplication of data in tables in case of multiple subclasses object
- Define Type
 - ✓ Create or replace type Person_tp as object (pid varchar2(10), pname varchar2(20), per_type varchar2(15)) NOT FINAL;
- Define Table
 - ✓ CREATE TABLE PersonP OF Person_TP (pid NOT NULL, per_type check ('student', 'staff'), PRIMARY KEY (pid));
- Create Student subtype


```
CREATE OR REPLACE TYPE Student_TP UNDER PERSON_TP
    (cgpa number(5,2),
    MEMBER PROCEDURE Insert_StudentP(
        new_id IN VARCHAR2,
        new_name IN VARCHAR2,
        cgpa IN number),
    MEMBER PROCEDURE Delete_StudentP)
```
- Define staff subtype


```
CREATE OR REPLACE TYPE Staff_TP UNDER PERSON_TP
    (job text,
    MEMBER PROCEDURE Insert_StaffP(
        new_id IN VARCHAR2,
        new_name IN VARCHAR2,
        job IN VARCHAR2),
    MEMBER PROCEDURE Delete_StaffP)
```

- // Create type body for Person
 CREATE OR REPLACE TYPE BODY Person_TUN AS
 MEMBER PROCEDURE Insert_Person(new_id IN VARCHAR2,
 new_name IN VARCHAR2) IS
 BEGIN
 INSERT INTO PersonU VALUES (new_id, new_name);
 END Insert_Person;
 MEMBER PROCEDURE Delete_Person IS
 BEGIN
 DELETE FROM PersonU
 WHERE PersonU.pid = self.pid;
 END Delete_Person;
 END;
- // Create type body for Student
 CREATE OR REPLACE TYPE BODY Student_TUN MEMBER PROCEDURE Insert_StudentP(new_id IN
 VARCHAR2, new_name IN VARCHAR2, cgpa IN number (5,2) IS
 BEGIN
 INSERT INTO PersonP VALUES
 (Student_TP(new_id, new_name, 'Student', cgpa);
 END Insert_StudentP;

 • MEMBER PROCEDURE Delete_StudentP
 BEGIN
 DELETE FROM PersonP
 WHERE PersonP.pid = self.id;
 END Delete_StudentP
 END;

Mutual Exclusion Inheritance with OO Approach

- An object can be of supertype.
- Or it can be anyone of the subtypes
- It cannot be of more than one subtypes.
- // Create type body for Person
 CREATE OR REPLACE TYPE BODY Person_TUN AS
 MEMBER PROCEDURE Insert_Person(new_id IN VARCHAR2,
 new_name IN VARCHAR2) IS
 BEGIN
 INSERT INTO PersonU VALUES (new_id, new_name);
 END Insert_Person;
 MEMBER PROCEDURE Delete_Person IS
 BEGIN

```
DELETE FROM PersonU
WHERE PersonU.pid = self.pid;
END Delete_Person;
```

```
END;
```

- // Create type body for Student
CREATE OR REPLACE TYPE BODY Student_TUN MEMBER PROCEDURE Insert_StudentP(new_id IN
VARCHAR2, new_name IN VARCHAR2, cgpa IN number)
BEGIN
INSERT INTO PersonP VALUES
(new_id, new_name, 'Student', cgpa);
END Insert_StudentP;

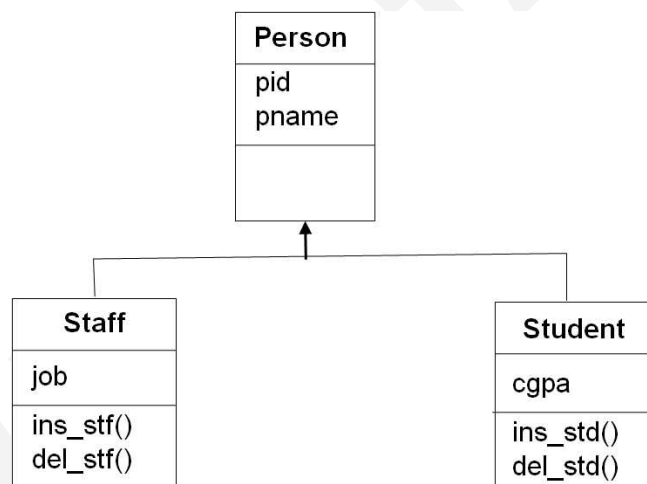
Lecture -32

Implementation of Generic Methods

-
-

Partition Inheritance

- Same as Mutual Exclusion EXCEPT that a super type object must be of one of the sub-types type.
- Implementation using Relational Approach:
 - ✓ Same as of Mutual Exclusion.
 - ✓ That is, we define types for super and sub classes.
 - ✓ All having id attribute in common and remaining individual ones.
 - ✓ Super type having a type attribute.
- Implementation using Relational Approach:
 - ✓ We define tables for all types.
 - ✓ Id attribute as PK in all tables.
 - ✓ Id attribute as FK in subtype tables.
 - ✓ Type attribute in supertype table contains “not null” constraint



Partition Inheritance – Example

- Define Type
Create or replace type Person_tp as object (pid varchar2(10), pname varchar2(20), per_type varchar2(15)) NOT FINAL;
No Insertion and Deletion Methods.
So no implementation of type body required.
- Define Table
CREATE TABLE PersonP OF Person_TP (pid NOT NULL, per_type check ('student', 'staff'), PRIMARY KEY (pid));

- Create Student subtype

```
CREATE OR REPLACE TYPE Student_TP UNDER PERSON_TP (cgpa IN number(5,2),
MEMBER PROCEDURE Insert_StudentP(
    new_id IN VARCHAR2, new_name IN VARCHAR2, cgpa IN number),
MEMBER PROCEDURE Delete_StudentP)
No Table to be defined
```
- Define staff subtype

```
CREATE OR REPLACE TYPE Staff_TP UNDER PERSON_TP (job text,
MEMBER PROCEDURE Insert_StaffP(
    new_id IN VARCHAR2,
    new_name IN VARCHAR2,
    job IN VARCHAR2),
MEMBER PROCEDURE Delete_StaffP)
No Table to be defined
```
- Type body of Student (without table)

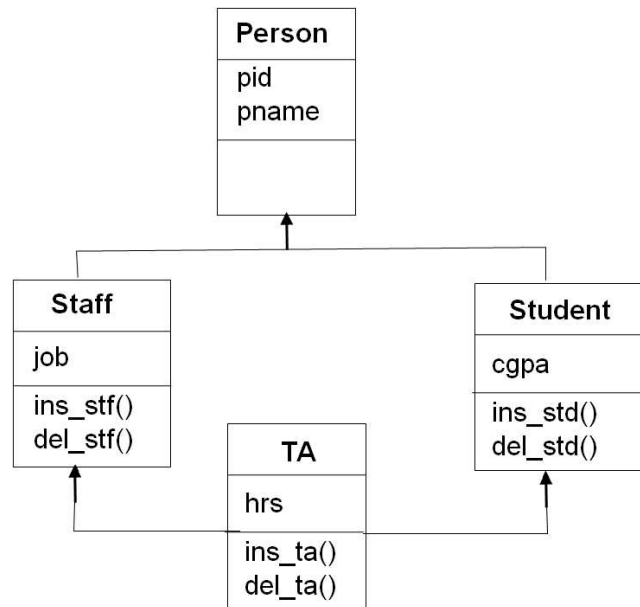
```
CREATE OR REPLACE TYPE BODY Student_TP MEMBER PROCEDURE Insert_StudentP(
    new_id IN VARCHAR2, new_name IN VARCHAR2, cgpa IN number (5,2) IS
BEGIN
    INSERT INTO PersonP VALUES (Student_TP
                                (new_id, new_name, 'Student', cgpa));
END Insert_StudentP;
```
- MEMBER PROCEDURE Delete_StudentP

```
BEGIN
    DELETE FROM PersonP
    WHERE PersonP.pid = self.id;
END Delete_StudentP
END;
```

Multiple Inheritance

- Not supported till OO Oracle till 9i.
- However, using relational features, it can be created.
- Define MI Type

```
CREATE OR REPLACE TA_tmi as object
    (pid varchar2(10), hrs number)
MEMBER PROCEDURE Insert_TAMI( new_id IN VARCHAR2, new_Name varchar2,
    new_job varchar2, new_cgpa number (5,2), hrs IN number),
MEMBER PROCEDURE Delete_TAMI);
```



- Create Table
CREATE TABLE TAMI OF TA_tmi (pid NOT NULL, PRIMARY KEY (pid),
foreign key (pid) references PersonMI (pid) ON DELETE CASCADE);
- Define Type Body
CREATE OR REPLACE TYPE BODY TA_tmi
MEMBER PROCEDURE Insert_TAMI(new_id IN VARCHAR2, new_name IN VARCHAR2, new_cgpa
IN number (5,2), new_job varchar2, new_hrs number) IS
BEGIN
INSERT INTO PersonMI VALUES (new_id, new_name);
INSERT INTO StudentMI VALUES (new_id, new_cgpa);
INSERT INTO StaffMI VALUES (new_id, new_job);
INSERT INTO TAMI VALUES (new_id, new_hrs);
END Insert_TAMI;
- MEMBER PROCEDURE Delete_TAMI
BEGIN
DELETE FROM PersonMI
WHERE PersonMI.pid = self.id;
END Delete_TAMI
END;

Lecture -33**OO Oracle**

-

Inheritance

-

User-Defined Queries

- Queries that are defined to access data from objects linked through different OO relationships, like inheritance, aggregation, association etc.
- Two main types:
 - ✓ Subclass queries
 - ✓ Superclass queries

Subclass Queries

- Queries defined in inheritance hierarchy.
- Query access data from subclass(es) where the selection predicate is originated at the superclass.
- In case of multiple tables (Relational Approach), a join is involved based on the common id attribute.
Select s.cgpa from student s, person p where s.pid = p.pid and p.name = 'Walli Khan'
- In case of Inheritance implemented using OO Oracle features, we use "treat".
SELECT TREAT(VALUE (p) as StudentT).cgpa from Person p where p.pName = 'Walli Khan'

OODBMS

- Objectivity/DB
- Ref: Objectivity/C++ Programmer's Guide (Release 10.0)

Objectivity Components

- Written in a Prog Interface, like C++
 1. The processes involved in an
 - ✓ Federated DB (FDB) Services
 - ✓ Lock Server
 2. Transactions, the mechanism for organizing operations on O/Dbobjects
 - ✓ Begin, Commit/Abort (ACID & Checkpoints supported)

Object Types

-
-

Objectivity/DB Objects

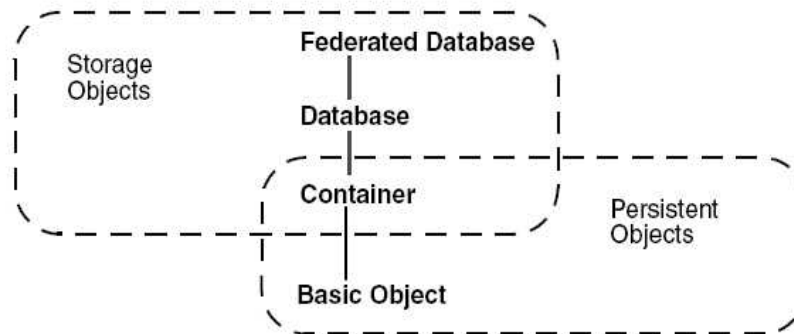
- Four types
 - ✓ Basic Object (defined by user in O/DB tools or the application interface).
- Container
 - ✓ Groups basic objects (physically clustered), efficient access.
 - ✓ Unit of locking (overall efficiency).
- Database
 - ✓ Contains app and sys created containers
 - ✓ Physically maintained as a file
 - ✓ Unit to distribute data physically
- Federated DB
 - ✓ Contains sys and app databases
 - ✓ Contains a system DB that contains global catalogue & schema
 - ✓ Schema is language independent
 - ✓ O/DB Objects arranged in containment hierarchy.
 - ✓ FDB at the top and basic object at the bottom.

Lecture -34

Objectivity/DB Object Types

1. Basic Object
2. Container
3. Database
4. Federated DB

Storage Hierarchy of O/DB



Partition Component

- Its an optional component.
- Every O/DB object has an *identifier*; All storage objects and autonomous partitions are given identifiers on creation.
- For a basic object is called its *object identifier or OID*; may change if object moved to other container; can be reused.

Working with O/DB objects

- FDB created through tool; remaining by application interface.
- 1. Find object in FDB; gets the OID of the object; through name or a link; 'getting an object'.
- 2. Object to application
 - ✓ Acquire appropriate lock (R/U).
 - ✓ Obtain memory representation.

Locking O/DB Objects

- Objects have to be locked.
- Implicit (application interface) or explicit locking (reduces concurrency).
- Containers are actual unit of locking for persistent objects.
- Concurrent Access, different containers by different Transactions, or same container if compatible.

Persistence Capable Classes

- Classes that have persistence behavior, they can be stored in O/DB persistently.
- Instantiation can create transient objects.

- PCC support transient only objects.
- However, certain constraints make them embeddable.

Persistence-Collection Classes

- Persistence-collection classes
- An instance called persistent object.
- Sets, lists, maps, name-maps.

O/DB Data Model

- A persistence-capable class has *attributes and associations*.
- Attributes' types supported: primitive, reference, embedded-class types.
- Association: links source and destination.
- Can be 'to-one' or 'to-many' types; like ref but better.
- Maintains reverse; manually or automatically.
- Action to be taken if associated object deleted.

Operations on O/DB Objects

- Creating objects

Objectivity/DB Object	Create by Tool	Create by Application
Federated Database	Yes; oonewfd or Assist	No
Database	Yes; oonewdb or Assist	Yes
Container	Yes; Assist	Yes
Basic Object	Yes; Assist	Yes

Linking objects together

- Preparing objects for lookup
 - ✓ Give container a name then search into it.
 - ✓ Persistent object can get scope name that can be searched later.
 - ✓ Place object in name-map or object-map.

Finding objects

- Using system name of container or database.
- Usually by looking up the containing object.
- Once an object found, linked object can also be following links.
- Predicate scan supported; scans containing object; time consuming; indexes supported.

Modifying Persistent Objects

- Modifying Persistent Objects:
 - ✓ Changes to position is to memory till transaction is committed.
 - ✓ O/DB has to be told so that no inconsistency occurs.
 - ✓ Each PI has its own mechanism to inform O/DB.
- Deleting

Deleting Objects

- By a tool or Application

Kind of Object	Delete by Tool	Delete by Application
Federated Database	Yes; oodeletefd	No
Database	Yes; oodeletedb or Assist	Yes
Container	Yes; Assist	Yes
Basic Object	Yes; Assist	Yes

Operations on O/DB Objects

- Finalized on commit or checkpoint.
- Non-garbage-collectible container; like in C++.
- Garbage-collectible container; like Java or Smalltalk.

Developing an O/DB Application

- Designing application
 - ✓ Identifying and designing classes
 - ✓ Organizing Persistent Objects
- Implementing and deploying
- Evolution of design

Evolving Classes

- Classes need to be modified due to changing requirements.
- During testing phase is easy.
- After deployment not that simple
 - ✓ Schema Evolution.
 - ✓ Object Conversion.

Lecture -35**Objectivity/DB**

- Reference: Objectivity/C++ Programmer's Guide, Release 10.0

Objectivity/C++ PI

- Targets of this chapter are:
 - ✓ **O/C++ PI**
 - ✓ Structure of an O/C++ application
 - ✓ Steps to develop an application
- O/C++ PI helps to build an application and manage interaction b/w application and O/DB.
- Consists of global classes, types etc defined in DLL.
- Application Control
 - ✓ Class ooObjy provides static methods for interaction.
- Application Control
 - ✓ Other two kinds of application object used are:
 - Connection objects and sessions
 - Startup properties, connection properties and session ones
 - Both are transient
- Connection object; instance of ooConnection; single connection object in one application; however we can also have multiple ones.
 - ✓ ooObjy is a factory for connection object.
- Session: an instance of ooSession.
 - ✓ Connection object serves as factory for session.
 - ✓ Controls interaction between app and connected FDB through transaction.
 - ✓ A session can have any number of transactions; one at a time.

Storage Objects

Class	Represents
ooFDObj	Federated database
ooDBObj	Database
ooContObj	Container (non-garbage-collectible)
ooGCContObj	Container (garbage-collectible)

- Basic Objects: some built-in PC, user defined PC classes derived from ooObjy; get persistent behavior.

Handles

- O/C++ application refers an O/DB object through a handle.
- Instance of ooHandle(cName), e.g, ooHandle(Person) pH;
- Like pointer; to access: pH.raiseSal(perc) or pH→raiseSal(perc)

Memory Management

- Handle stores OID and state about memory management.
- Open handle contains in-memory pointer (OID) of object.
- Closed handle contains disk based OID.
- Object Reference: a wrapper for OID of object; instance of ooRef(cN) class, e.g.

```
class Dept;
class Person : public ooObjY
{...
    ooRef(Dep) dept;
...}
```
- OR and OH can be used to access members of objects; not totally interchangeable; short ref also available.

Objectivity/C++ PI

- Object Iterator: a transient object for iterating over a group of objects; other types also available; iterates through iteration set; object of ooItr(CN).
- Exception classes: to handle error conditions; derived from ooException class; use of try and catch blocks.

Common Types & Constants

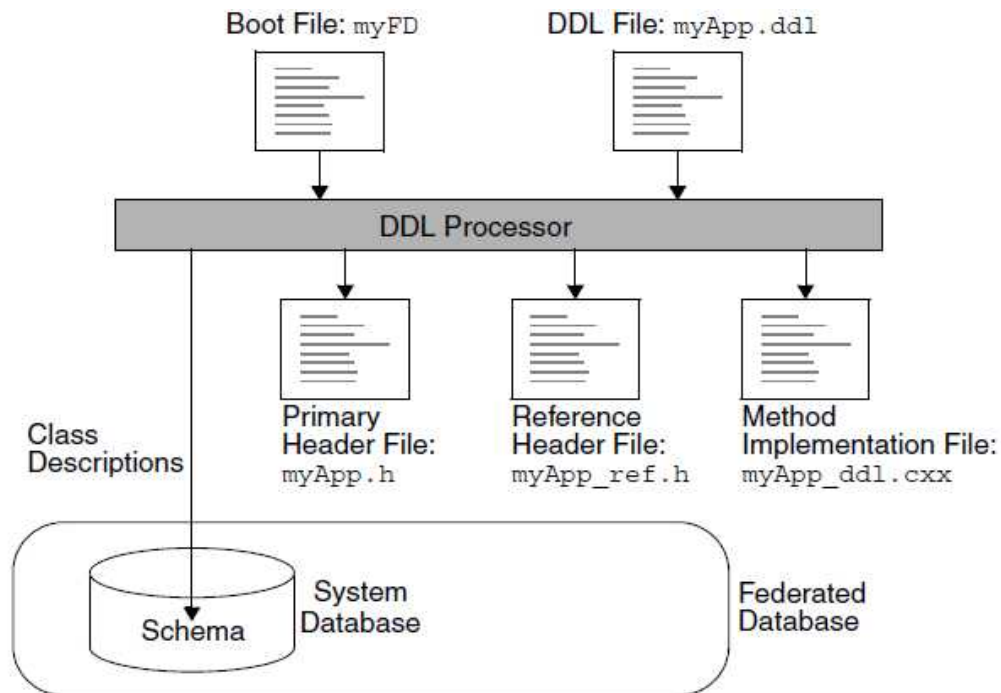
- Same as C++
- ooInt32, ooInt16, ooUInt16, ooFloat64, ooBoolean, ooCTrue, ooCFalse
- Access Modes: ooMode, oocRead, oocUpdate
- Status Code: oocSuccess, oocError

O/C++ App Development

- Creating fdb using oonewfd or Assist; creates a boot file, schema and global catalogue
- Defining PC classes: DDL used, like .h files, extension is .ddl
- Adding Class Descriptions to Schema

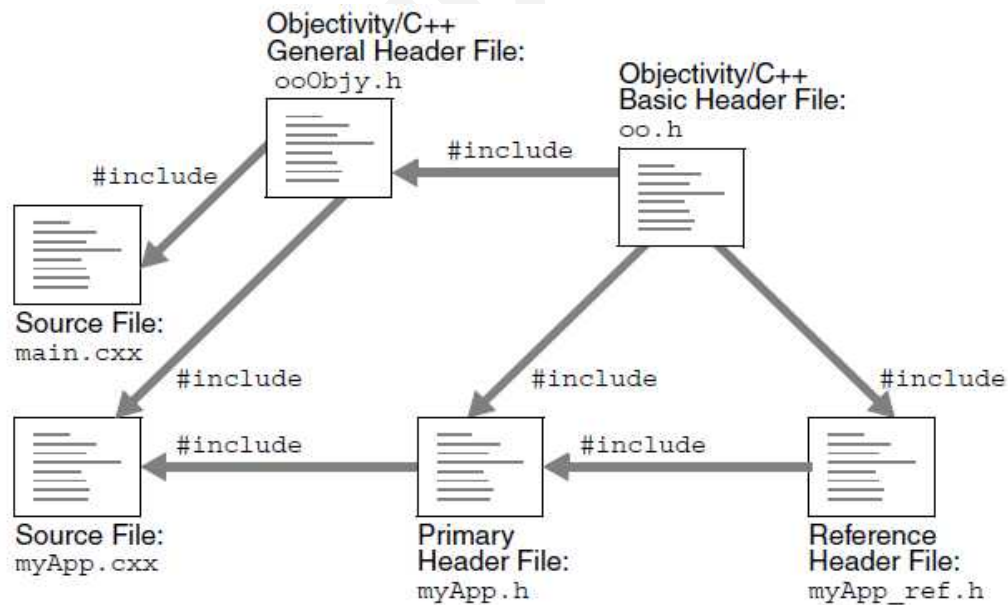
O/C++ App Development

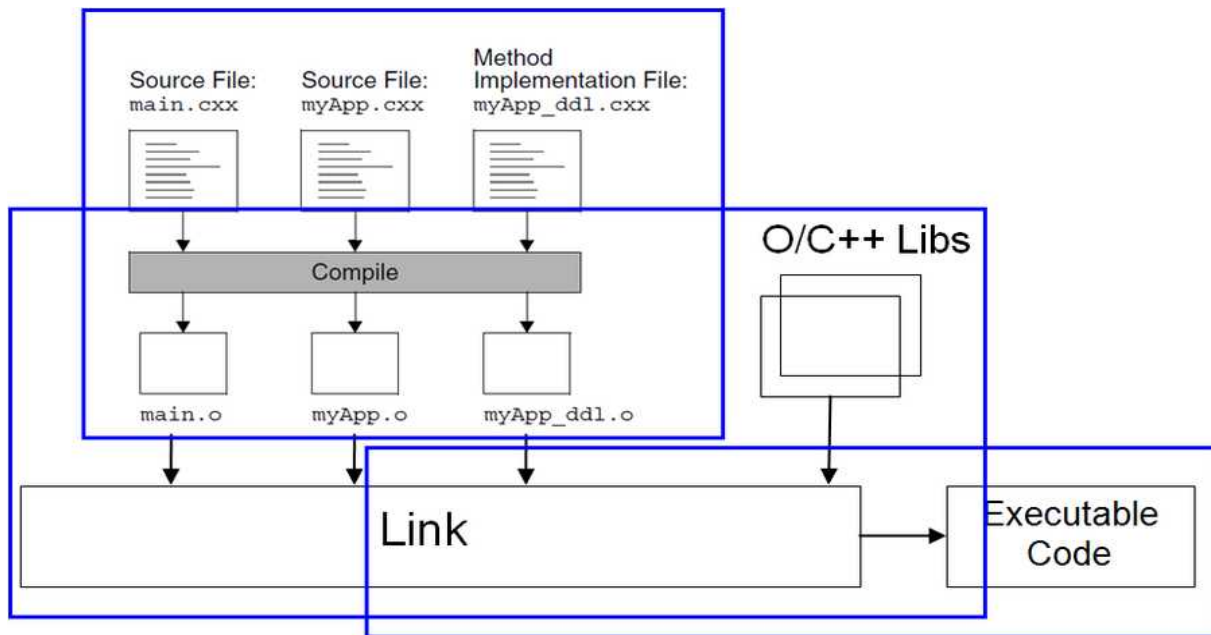
- DDL preprocessor processes .ddl
 - ✓ A primary header file (cDef.h).
 - ✓ A reference header file (cDef_ref.h); contains definitions of ooHandle(cN), ooRef(cN), ooItr(cN).
 - ✓ A method implementation file.



Developing App Source Code

- O/C++ PI used to build application, can use PC classes, and perform DB operations.



Compiling & Linking**Schema Evolution And Object Conversion**

- In case of schema change ddl processor has to be involved.
- Objects conversion is also required if objects created.

Structure of an O/C++ App

- Initialize and terminate interactions: using `ooObjy::startup` & `ooObjy::shutdown` static methods.
- // Application code file

```
#include <ooObjy.h>
int rv = 0; ...
int main(const int argc, const char *const argv[]) {
... try {
    ooObjy::startup();
    ....}
    catch (ooException &error) {
        ... returnVal = 1;
    }
    catch(...) {
        returnVal = 1;
    }
}
```
- // Terminate interaction with Objectivity/DB

```
ooObjy::shutdown(); .....
return rv;
}
```

O/C++ Application

- Creating Application Objects
- Establish connection b/w application program and FDB; indicates application intends to interact with FDB; then create session to interact.
- Beginning and Ending Transactions: session object is used to initiate transaction; guarantees consistency.
- // Application code file

```
#include <ooObjy.h>
void dbOp (arg list) {
    ---
    ooConnection *connection = ooObjy::getConnection("ourfdb.boot");
    ooSession *session = connection->createSession("oursession");
    session->begin(ooUpdate); // transaction begins
    ... // Perform Objectivity/DB operations
    session->commit(); // Commit transaction
    ...
}
```

Lecture -36

Programming Language Interface (C++)

•

Creating and Finding Objects

- // Application code file

```
#include <ooObjy.h>
ooConnection *connection = ooObjy::getConnection("ourFDB.boot");
ooHandle(ooDBObj) dbH; // DB handle
ooSession *session = ..... // Session
session->begin(oocUpdate);
if (!dbH.exist(session->fd(), "ourDB"))
dbH = new ooDBObj("ourDB");
session->commit();
...
```

Accessing Persistent Objects

- We can do through handle.
- Can get and set attributes.
- ddl processor generates methods to set associations.

```
#include <ooObjy.h>
#include "ourexample.h"
... ooSession *session = ... ;
session->begin(oocUpdate); // Tr with update
ooHandle(Staff) bossH = ... ;
ooHandle(Staff) assistantH = ... ;
cout << assistantH->name << endl;
assistantH->set_manager(bossH);
bossH.update();
(bossH->numSupervised)++;
session->commit();
```

Deleting an Object

- // Application code file

```
#include <ooObjy.h>
#include "ourexample.h"
... ooSession *session = ... ;
session->begin(oocUpdate); // Begin a transaction
ooHandle(Staff) retiredH = ... ; // Get a handle
ooDelete(retiredH); // Delete the Staff
session->commit(); // Commit the transaction
```

Objectivity/DB

- Reference: Objectivity/C++ Programmer's Guide, Release 10.0 (Chapter 3)

O/C++ Processes

- O/C++ classes and mechanisms
- Basic Interaction
 1. Initialize interaction with O/DB
 2. A logical connection to an FDB
 3. Create one or more sessions
 4. Use sessions to perform ops
 5. Terminate interaction with O/DB

Logical and Physical Connections

-

1- Initializing Interaction with O/DB

- ooObjy::startup() // Single thread
- ooObjy::initializeThread // Multi thread
- Setting startup properties (*affect application globally*)
- Max file descriptors in a session (12 by default)

Making a logical connection

- ooObjy::getConnection
- ooConnection *connection1 = ooObjy::getConnection("FDB1.boot");
- ooConnection *connection2 = ooObjy::getConnection("FDB2.boot");
- ooSession* session1= connection1->createSession("First Session");
.....
- Session1->detach();
.....
- ooSession* session2= connection2->createSession("Second Session");
.....

Making Multiple Logical Connections

- Usually one FDB.
- For multiple, we need multiple connections.

Controlling Automatic Recovery

- Connection object created with Automatic Recovery 'on'
- If Application Fails
- If lock server fails
- Terminating Interaction with Objectivity/DB
- ooObjy::shutdown // static method

2- Sessions

- An instance of class ooSession
 - ✓ Individual or pooled
 - ✓ Connection object used as factory
- Transaction Services
 - ✓ Operations performed in the form of transaction; session can have multiple; one active at a time
 - ✓ A session performs a series of tasks

Operating Environment

- Session operates in an OE
- Session resources include
 - ✓ An O/DB Cache
 - ✓ Number of file descriptors
 - ✓ A message-handler function
 - ✓ A two-machine handler function.

Session Properties

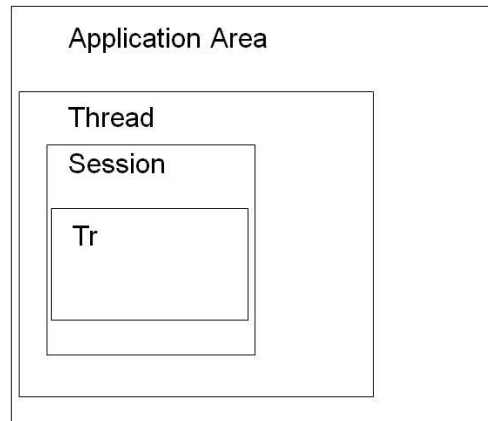
- Govern its operating environment; can be classified as:
 - ✓ Set to default or application specific values when the session is created.
 - ✓ Initialized to default values when the session is created.
 - ✓ Reset to default values each time the session performs a transaction.

Lecture -37

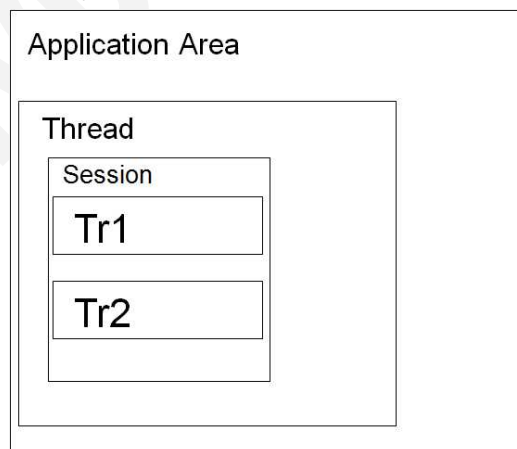
Objectivity/DB

Sessions

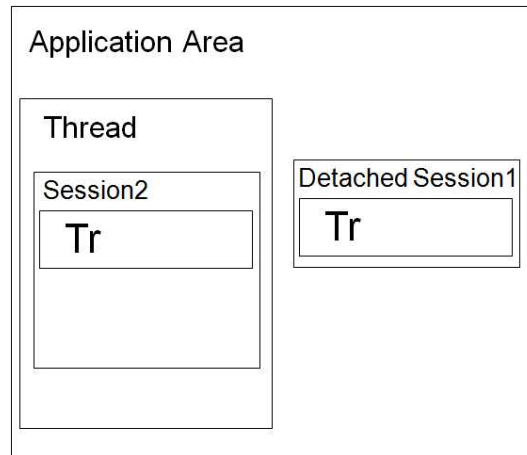
- Related Objects
 - ✓ Connection, FDB, Clustering Strategy.
 - ✓ Additional: Transaction, Log, Event listeners.
- Single Thread, Single Session, Single Transaction.
- Single Thread, Single Session, Single Transaction



- Animate all at once (in sequence but no separate steps needed) as displayed in slide the colors are referred by professor, so they must be similar (you can do dark or dim upon your own choice but basic color should be same)
- Single Thread, Single Session, Series of Transaction



- Single Thread, Multiple Session, Single Transaction



- Attached & Detached Sessions
- Handles and Object References

Creating Individual Sessions

- Use of createsession of connection object.
 - Named and unnamed.
- ```
void printInventory() {
 ooSession *session = 0;
 try {
 ooConnection *connection = ... ;
 // Create the session
 session = connection->createSession(0);
 ... // Look up information and print report }
 catch (ooException &error) {
 cerr << error.what() << endl;
 returnVal = 1; }
 catch(...) {
 cerr << "unknown error" << endl;
 int returnVal = 1; }
 // Delete the session
 delete session;
}
```

### Session Pool

- Object created by connection object maintains multiple similar session properties:
  - ✓ Name
  - ✓ Hard and soft limits
  - ✓ Session-waiting policy
  - ✓ Fixed on creation

- Using session pool to limit concurrency.
- For example, consider a server application that handles requests from client applications. The server application creates one thread for each client; each thread obtains a session, when needed, from a session pool.
- When a client application submits request, corresponding thread in server application obtains a session, processes request in a transaction, returns session to the pool. If session pool's hard limit is 6, it will contain a maximum of 6 sessions; at most 6 threads can perform transactions simultaneously.
- Server application could still service more than 6 clients simultaneously. If threads for 6 clients are in active transactions when a new thread requests a session, latter thread will have to wait until one of the former threads finishes its transaction and returns its session to pool.

### Creating a Session Pool

- ```
#include <ooObjy.h>
ooConnection *connection = ... ;
// Create a session pool
connection->createSessionPool(
    "MyPool", // Name of session pool
    10,       // Soft limit
    14);      // Hard limit
```
- ```
#include <ooObjy.h>
...
// Request a session from MyPool
ooSession *mysession =
connection->getSessionFromPool("MyPool");
```

### Selection of Pooled Session

- If pool has a single available session, getSessionFromPool returns it.
- If pool has multiple available session, getSessionFromPool chooses session that was returned to the pool most recently.
- If session pool has no available session, has created less than the hard limit of sessions, new session is created with property values specified when session pool was created, getSessionFromPool returns the new session.
- If session pool has no available session, has already created its maximum number of sessions, getSessionFromPool behaves according to the session pool's session-waiting policy.

**Waiting for a Session**

- oocNoWait, oocWait n (seconds)
  - ooTooManySessions (exception)
- ```
#include <ooObjy.h>

...

void addDeposit(const char *customerID, ooUInt32 amount) {
    ooConnection *connection = ... ;
    ooSession *mysession =
    connection->getSessionFromPool(
    "AccountsPool", customerID);
    ...
    session->returnSessionToPool();
}
```

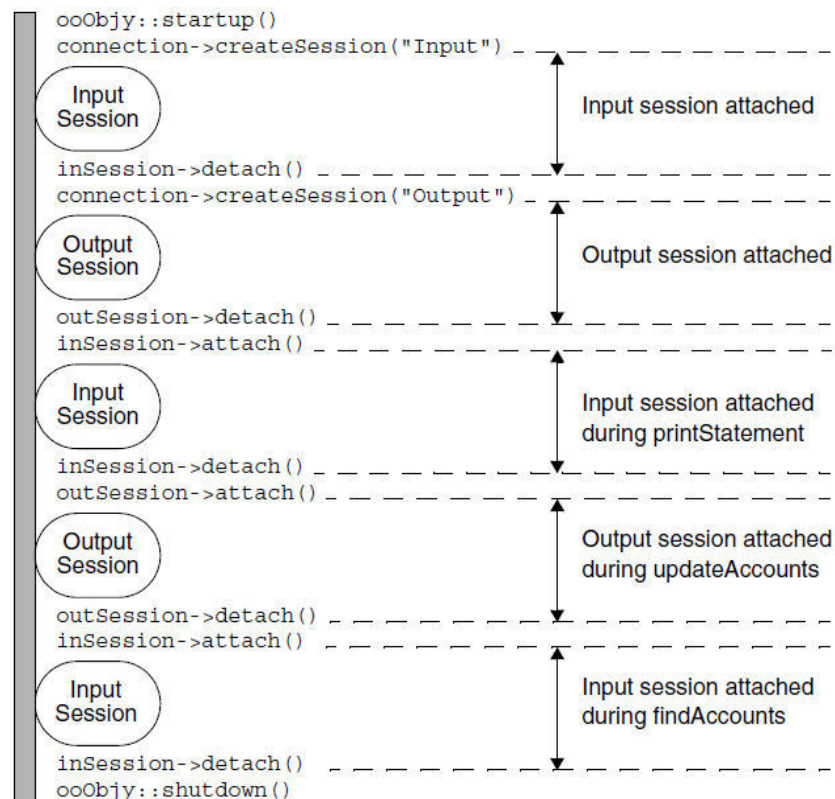
Sessions in a Multithreaded Application

- Main thread
 - ✓ Starts with ooObjy::starup()
 - ✓ Ends with ooObjy::shutdown();
 - Nonmain thread
 - ✓ Created ooObjy::initializeThread
 - ✓ Ends ooObjy::terminateThread
- ```
int main(const int argc, const char *const argv[]) {
 int returnVal = 0;
 try {
 // Initialize main thread's interaction
 ooObjy::startup();
 performObjectivityOperations(argc, argv);
 }
 catch (ooException &error) {
 cerr << error.what() << endl;
 returnVal = 1;
 }
 catch (...) {
 cerr << "unknown error" << endl;
 returnVal = 1;
 }
 ooObjy::shutdown();
 return returnVal;
}
```

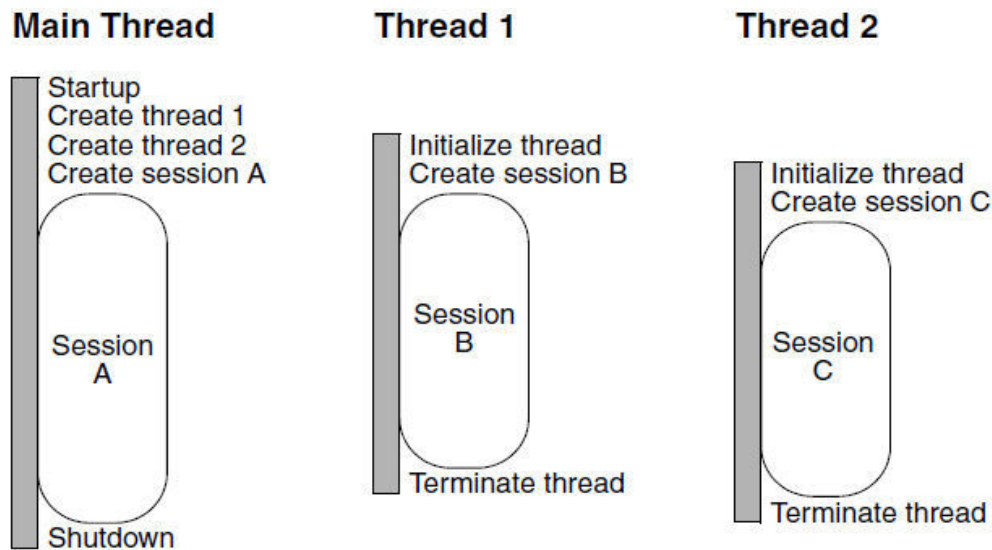
- // Function that performs Objectivity/DB operations,  
 void performObjectivityOperations(  
   const int argc, const char \*const argv[]) {  
   ...  
   // Create a thread that calls threadFn1  
   createThread (...,&threadFn1,...,parameters,...);  
   // Pseudocode  
   ...  
   // Create a thread that calls threadFn2  
   createThread (...,&threadFn2,...,parameters,...);  
   // Pseudocode  
   ...  
 }

### Session Usage Model

- Primary Restriction
  - ✓ One thread one session at a time
  - ✓ One session in one thread at a time
- Different models
  - One thread multiple session
    - Attach/Deattach methods



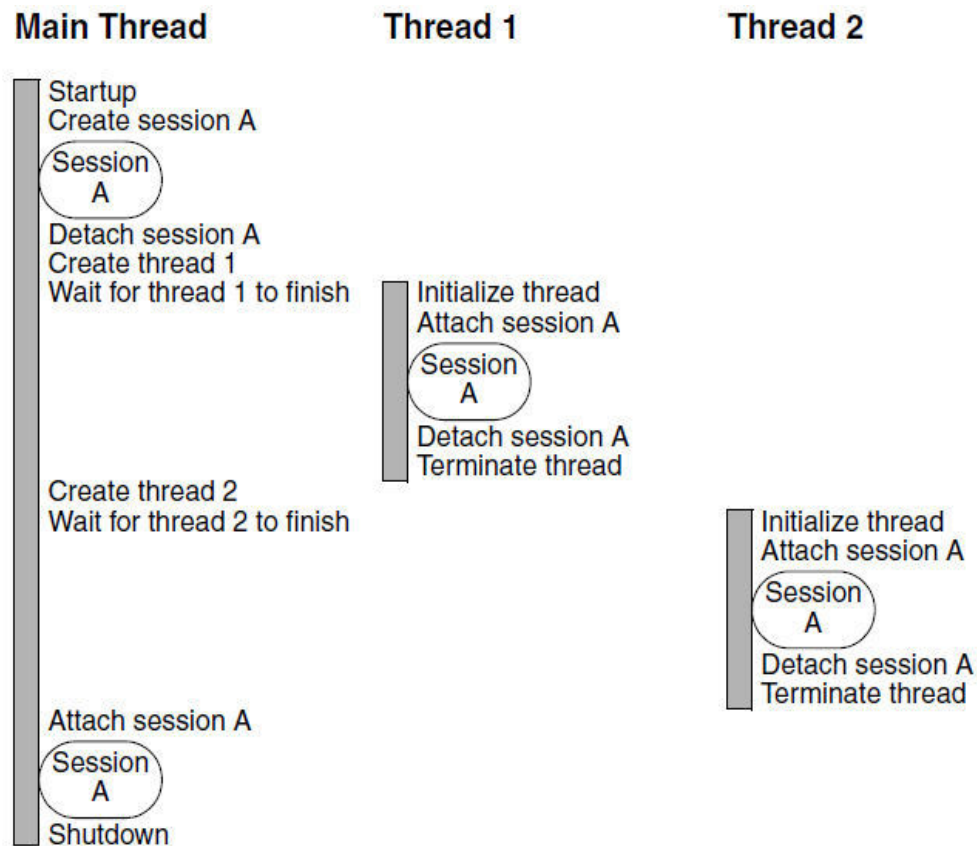
- 2- Multiple threads, each with
- its own session



Try to adjust without animation or zooming Otherwise zoom in 2 steps splitted by line. Line is just to help you (no need to draw it in actual slide); Text is:

- Main Thread
  - Startup
  - Create thread 1
  - Create thread 2
  - Create session A
  - Session A
  - Shutdown
- Thread 1
  - Initialize thread
  - Create session B
  - Session B
  - Terminate thread
- Thread 2
  - Initialize thread
  - Create session C
  - Session C
  - Terminate thread

## 3- Multiple Threads sharing a session



Zoom in steps as mentioned in slide show the boxes are just to help you, they are not needed in actual slide; Text is:

Main Thread  
Startup  
Create session A  
Session A  
Detach session A  
Create thread 1  
Wait for thread 1 to finish

Thread 1  
Initialize thread  
Attach session A  
Session A  
Detach session A  
Terminate thread

Create thread 2  
Wait for thread 2 to finish  
Attach session A  
Session A  
Shutdown

Thread 2  
Initialize thread  
Attach session A  
Session A  
Detach session A  
Terminate thread

**Lecture -38****Transactions**

- “A sub-section of an application operations during which the application interacts with FDB”.
- 
- 

**Objectivity/DB Transaction**

- Takes and Leaves a database in consistent states.
- Extent of Transaction.
- Indivisible Unit of Work.

**Transactions**

- **Locking Mechanism**

- ✓
- ✓
- ✓
- ✓

- **Concurrent Access**

- ✓
- ✓
- ✓
- ✓
- ✓

- **Granularity**

- ✓
- ✓
- ✓
- ✓

**Lecture -39****Multiple Transactions**

- 
- 

**Transactions Mode**

- 
- 

**Transactions Locks**

- 
- 

**Journal File**

- 
- 

**Transaction Mechanism**

- Session method: begin();
- Active
- Open mode
- MROW mode
- Index mode

**Transaction Mechanism**

- oocRead-Begin a read transaction.
- oocUpdate-Begin an update transaction.
- Exception Handling

```
#include <ooObjy.h>
...
ooSession *session = ... ;
session->begin(oocRead); // Begin read transaction
...
session->commit(); // Commit the transaction
...
session->begin(oocUpdate); // Begin update transaction
...
session->commit(); // Commit the transaction
```

**Committing Transaction**

- Indicates a successful end to the current indivisible unit of work.
- Saves all newly created or modified Objectivity/DB objects to the FDB.
- Removes any deleted persistent objects from the federated database.
- Updates applicable indexes according to session's index mode.
- Ends transaction, changes session's transaction open mode to indicate session is not accessing FDB.
- Closes all handles. The closed handles retain object identifiers of objects to which they refer.
- Terminates iteration of all object iterators.
- Closes all open persistent objects, but leaves their memory representations in the cache.
- Releases any locks acquired in the course of the transaction.
- May, but need not, close the files accessed by the transaction.

```
#include <ooObjy.h>
...
ooSession *session = ... ;
session->begin(oocRead); // Begin read transaction
... // Read persistent objects
session->commit(); // Commit the transaction
```

**Committing Transaction**

- 
- 

**Transaction Mechanism**

- 
- 

**Aborting Transaction**

- oocHandleToNull-Set open handles to null; replace their object identifiers with null.
- oocHandleToOID-Close open handles, which invalidates their pointers, but preserves the object identifiers they contain.
- Indicates unsuccessful end to current indivisible unit of work.
- Rolls back changes, leaving FDB in logical state it was in before current unit of work began.
- Ends transaction, changes session's transaction open mode to indicate session is not in a transaction.
- Closes all handles/sets them: null
- Terminates the iteration of all object iterators.
- Closes all open persistent objects, and removes their memory representations from cache.
- Releases any locks acquired in course of transaction.
- Closes all files accessed by transaction.

```

#include <ooObjy.h>
...
ooSession *session = ... ;
session->begin(oocUpdate); //Begin update transaction
... //Update persistent objects
if (someCondition) //Test some condition
 session->abort(); //True:abort the transaction
else
 session->commit(); //Otherwise, commit

```

### Checkpointing

- Successfully ends current indivisible unit of work.
- Saves all newly created or modified Objectivity/DB objects to FDB.
- Physically removes any deleted persistent objects from FDB.
- Updates all applicable indexes according to session's index mode.
- Begins a new indivisible unit of work.
- *Does not* end the transaction.
- *Does not* close persistent objects.
- *Does not* close handles.
- *Does not* terminate iteration of any object iterators.
- *Does not* close files accessed by transaction.
- Retains same locks, preventing other transactions.

```

• #include <ooObjy.h>
...
ooSession *session = ... ;
session->begin(oocUpdate); //Begin update transaction
... //Access persistent objects
session->checkpoint(); //Checkpoint transaction
... //Access persistent objects
session->commit(); //Commit transaction

```

### Downgrading Locks

- 
- 
-

**Nested Calls to Begin**

```
#include <ooObjy.h>
void payTaxes(ooSession *session) {
 session->begin(oocUpdate);
 session->commit(); }
void buyHouse(ooSession *session) {
 session->begin(oocUpdate);
 payTaxes(); // This call results in nesting
 session->commit(); }
void doSomething(ooSession *session) {
 payTaxes(session);
 buyHouse(session); }
```

**Upgrade Transaction**

```
#include <ooObjy.h>
void readSomething(ooSession *session) {
 session->begin(oocRead);
 if (someCondition)
 changeSomething(session);
 session->commit();
}
void changeSomething(ooSession *session) {
 session->begin(oocUpdate);
 session->commit();
}
```

**Transaction Usage Guidelines**

- Keep important transactions short.
- Use short transactions for update-intensive objects.
- Use long transactions when slow network access is involved.

## Lecture -40

## OODBMS (Objectivity/DB)

## Schema Development

- 
- 

## Reference

- Objectivity/C++ Data Definition Language Release 10.0 (Chapter 1)
- 

## Schema Development

- Persistence-Capable Classes (PCC)
  - ✓ To be added to FDB schema.
- PC Classes
  - ✓ Derive directly or indirectly from ooObj (O/C++) class.
  - ✓ Non PCC: transient objects, can be stored inside a PCC object.
- Using O/DDL
  - ✓ Extension of C++; for PCC definition, other features: associations; similar syntax.
  - ✓ Class definitions placed in .ddl files.
  - ✓ Schema created through ddl processor; extracts type info from ddl files; creates schema in FDB.
  - ✓ Creates C++ files that contains
    - Extended PCC Classes
    - Relevant system-defined classes
  - ✓ Definitions can be modified; versioned if existing objects to be retained.
- Creating DDL files
  - ✓ Resemble .h files; can't be compiled directly; pre-processing
  - ✓ Basic DDL File Contents
- Adapting Existing .h files
  - ✓ Change the extension
  - ✓ Adapt class definition: Place appropriate base classes. Adjust data types, associations, pointers etc.
- `class Fleet; // Forward declaration`
- `class Vehicle {`
  - `public:`
  - `char *license;`
  - `char *type;`
  - `int doors;`
  - `int transmission;`
  - `bool available;`
  - `Fleet *inFleet;`
- `};`

- `class Car : public Vehicle ... {`  
`...`  
`};`
- `class Fleet; // Forward declaration`
- `class Vehicle: public ooObj {`  
`public:`  
`ooVString license;`  
`ooVString type;`  
`ooInt16 doors;`  
`ooInt8 transmission;`  
`ooBoolean available;`  
`ooRef(Fleet) inFleet <-> vehicles[ ]; // association`  
`};`
- `class Car : public Vehicle ... {`  
`... // from Vehicle`  
`};`

### Processing DDL Files

- DDL processor creates or modifies schema in FDB; make sure that:
  - ✓ FDB exists
  - ✓ Lock server started
  - ✓ “bin” folder exists that contains the ddl processor
  - ✓ ooddlx.exe is the processor file
- Files generated
  - ✓ A primary header file: `fname.h`
    - Contain additional members and methods from defined in PCC.
- Files generated
  - ✓ A reference (secondary) header file: `fname_ref.h`
    - Contains generated definitions of parameterized classes that support referencing and iterating over persistent objects.
- Files generated
  - ✓ An method implementation file `fname.cpp`
- // DDL file: `a.ddl`  
`class A : public ooObj { ...`  
`};`

- // Application code: main.cxx  

```
#include <a.h> // definition of A; includes a_ref.h
{ ...
 ooHandle(A) aH; // Requires definition of ooHandle(A)
 aH = new() A; // Requires definition of A
 ...
};
```

### User-Created Files

**a.ddl**

```
class A : public ooObj {
 ...
 void print();
};
```

### DDL-Generated Files

**a\_ref.h**

```
#ifndef ...
#include <oo.h>
#endif
...
class ooRef(A) : public ooRef(ooObj) {
 ...
class ooHandle(A) :
 public ooHandle(ooObj) { ...
```

**a.h**

```
#ifndef ...
#include <oo.h>
...
#include "a_ref.h"
#endif
class A : public ooObj {
 ...
 void print();
 ooHandle(A) ooThis(...) const;
 void *operator new(...);
 ...
};
```

**a.cxx**

```
#include "a.h"
/*methods on A */
void A::print() {
 ...
}
```

**main.cxx**

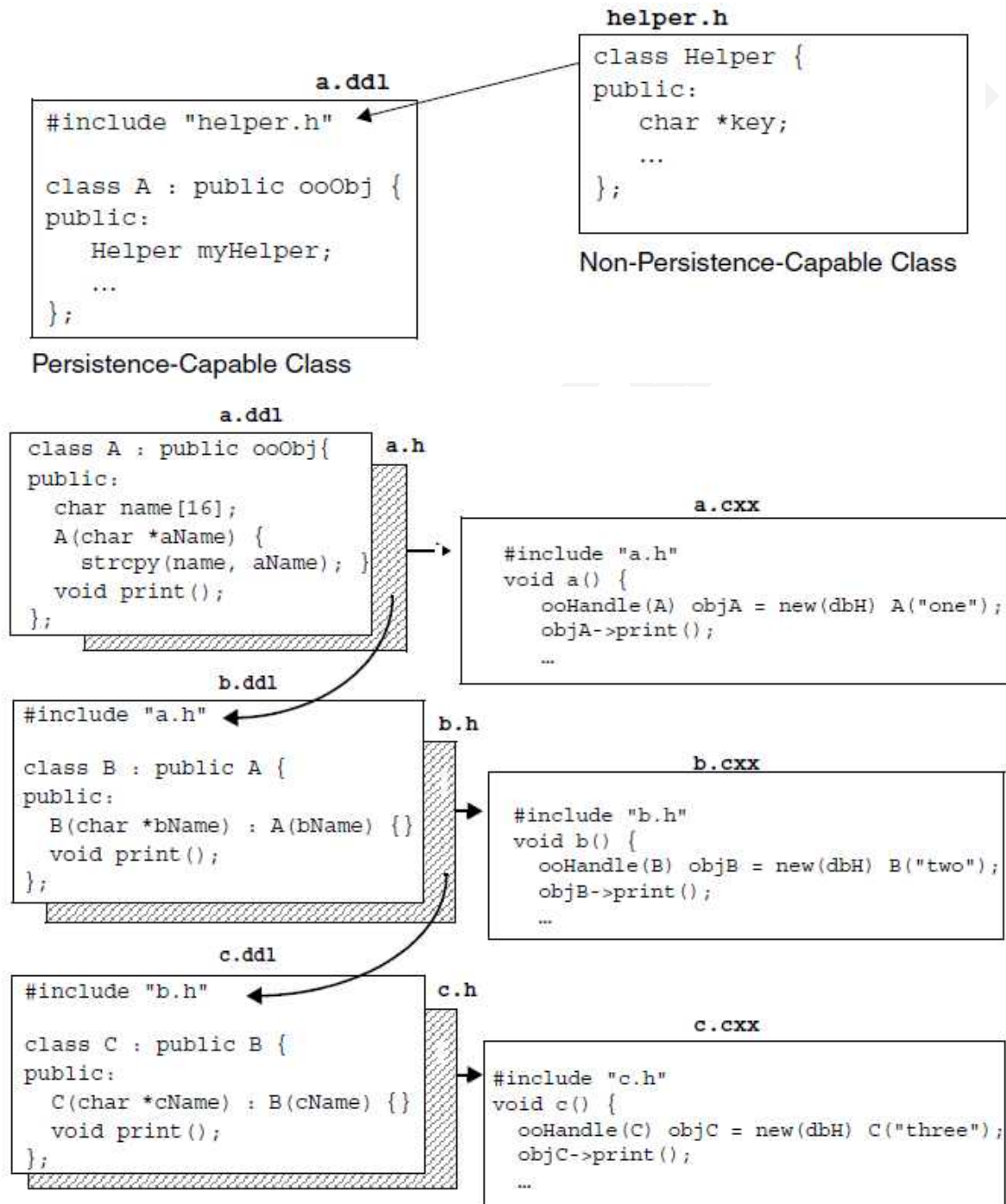
```
#include "a.h"
...
{
 ooHandle(A) aH;
 aH = new() A;
}
```

**a\_ddl.cxx**

```
#include "a.h"
/* methods on ooRef(A) */
ooStatus ooRef(A)::lookupObj(...)
...
/* methods on ooHandle(A) */
ooStatus ooHandle(A)::copy(...)
...
/* methods added to A */
ooHandle(A) A::ooThis(...) const {
 void *A::operator new(...) { ...
```

**Processing DDL Files**

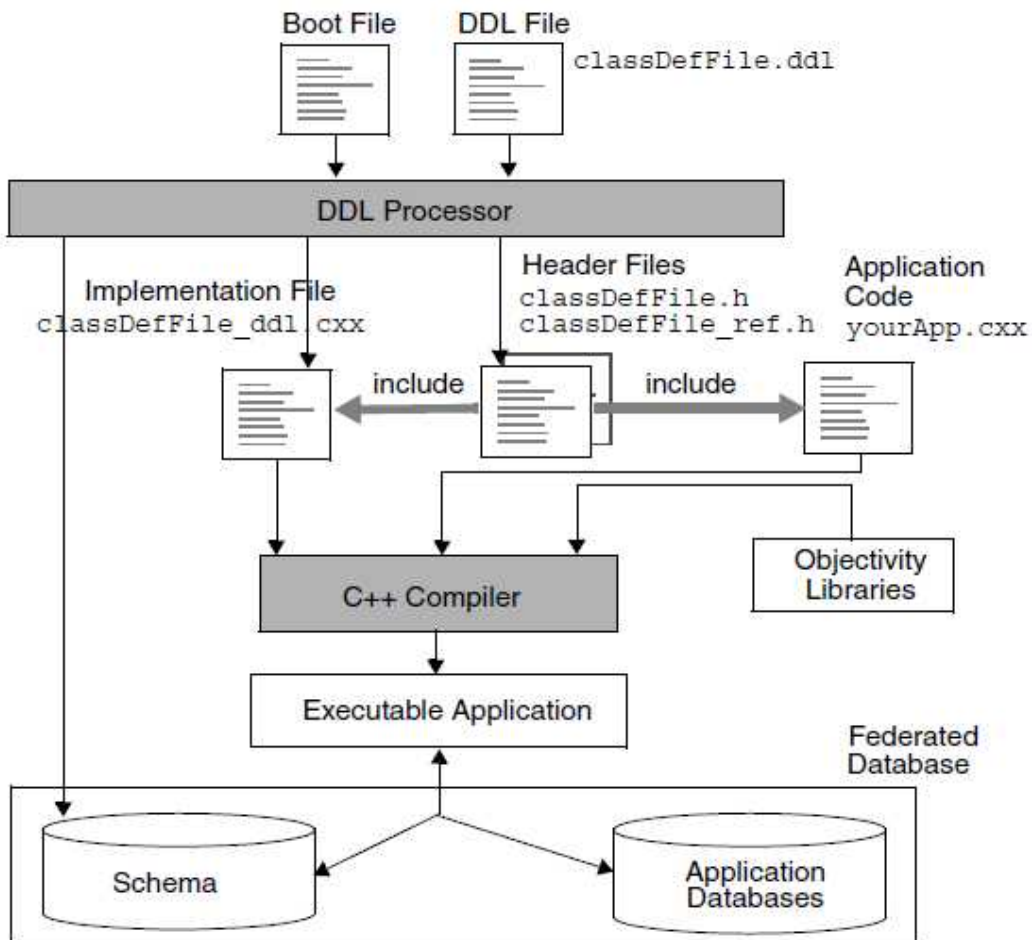
- Treatment of pre-processor directives.
- Dependencies on DDL-Generated code.
- Dividing definitions among multiple files.
  - ✓ Quite common in C++ programming.



**Processing DDL Files**

- If code in one DDL file references a PCC defined in a different DDL file, we have to arrange generated parameterized classes to be available too
- `#pragma ooclassref Book <book_ref.h>`
- `class Book;`  
`#pragma ooclassref Book <book_ref.h>`  
`class Library : public ooObj {`  
`ooRef(Book) books[ ] <-> theLibrary;`  
`...`  
`};`
- `class Library;`  
`#pragma ooclassref Library <library_ref.h>`  
`class Book : public ooObj {`  
`ooRef(Library) theLibrary <-> books[ ];`  
`...`  
`};`

## Lecture -41



## Defining PCCs

- Reference: Objectivity/C++ Data Definition Language Release 10.0 (Chapter 2)
- DDL used to define schema later created by DDL processor
- Extends C++ by:
  1. Implicitly including `oo.h`
  2. Implicitly defining parameterized classes
- DDL used to define schema later created by DDL processor
- Extends C++ by
  3. Provides syntax to define associations
  4. Provides `#pragma` directive
- Example
 

```
// DDL file
class Vehicle: public ooObj{ ... };
class Truck: public Vehicle { ... };
```

- Class Template can be made PCC

```
// DDL file
```

```
template <class T>
```

```
class Segment : public ooObj {
```

```
public:
```

```
 ooRef(T) toPointA : copy(delete);
```

```
 ooRef(T) toPointB : copy(delete);
```

```
};
```

```
// Instantiate the individual template classes
```

```
template class Segment<CartesianPoint>;
```

```
template class Segment<PolarPoint>;
```

### Component Data of a Class

- Attributes: constitute state of object, saved persistently in FDB.
- Associations: refer to objects of other classes.
- Attributes
- Same syntax as C++.
- O/C++ Primitive data types.

| Category                    | Type Name                                   | Alternate Name | ODMG Name | Description                  |
|-----------------------------|---------------------------------------------|----------------|-----------|------------------------------|
| Integer                     | ooInt8                                      | int8           | (None)    | 8-bit signed integer type    |
|                             | ooUInt8                                     | uint8          | d_Octet   | 8-bit unsigned integer type  |
|                             | ooInt16                                     | int16          | d_Short   | 16-bit signed integer type   |
|                             | ooUInt16                                    | uint16         | d_UShort  | 16-bit unsigned integer type |
|                             | ooInt32                                     | int32          | d_Long    | 32-bit signed integer type   |
|                             | ooUInt32                                    | uint32         | d_ULong   | 32-bit unsigned integer type |
|                             | ooInt64                                     | int64          | (None)    | 64-bit signed integer type   |
|                             | ooUInt64                                    | uint64         | (None)    | 64-bit unsigned integer type |
| Floating point <sup>a</sup> | ooFloat32                                   | float32        | d_Float   | 32-bit floating-point type   |
|                             | ooFloat64                                   | float64        | d_Double  | 64-bit floating-point type   |
| Character <sup>b</sup>      | ooChar                                      | char           | d_Char    | 8-bit character type         |
|                             | ooChar16                                    | (None)         | (None)    | 16-bit character type        |
|                             | ooChar32                                    | (None)         | (None)    | 32-bit character type        |
| Boolean <sup>b</sup>        | ooBoolean                                   | (None)         | d_Boolean | 8-bit true or false value    |
| Enumeration                 | Any Objectivity/C++-defined enumerated type |                |           | 32-bit signed integer type   |

**C++ Pointer Type**

- Useless in persistent objects; however can be used to refer to transient objects.
- Should be set to 0 before end of transaction.

**Object-reference Type**

- A smart pointer.
- Refers other objects using OID.
- Attribute type is ooRef.
- ooHandle(cName) cannot be used to refer object of other class type.

```
class Vehicle : public ooObj {
 ...
 ooRef(Fleet) fleet;
};
class Fleet: public ooObj {
 ...
 ooRef(Vehicle) vehicles[1000];
};
```

**Embedded-Class Types**

- Structure can be used, if:
  - ✓ It is non-PCC
  - ✓ Has no virtual base class
  - ✓ Contains no association
  - ✓ Has allowed data types

```
// DDL file
struct Point{
 ooInt32 xCoord;
 ooInt32 yCoord;
};
class Line : public ooObj {
 ...
 Point points[2];
 ...
};
```

**Other Supported Types**

- Enumerations
- String
- VArray
  - ✓ Same as C++ Array, size can vary at run time; ooVArrayT<eT>
  - ✓ *eT can be a primitive type, an object-reference, valid embedded-class; not VArrays*
- Date and Time

**Defining Associations**

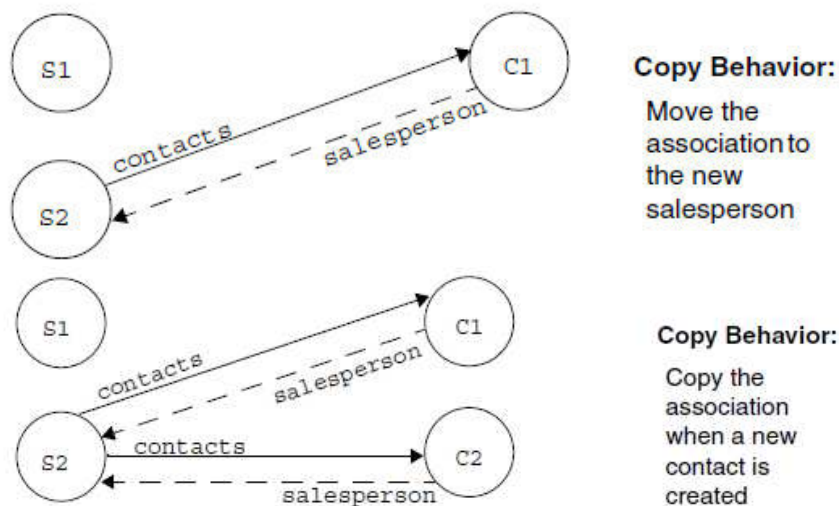
- **Reference: Objectivity/C++ Data Definition Language Release 10.0 (Chapter 3)**
- A property of a PCC; enables apps to link persistent instances of that class to each other or to persistent instances of some other class.
- One way to establish link is to use ooRef (clName); user responsible for modifications/changes.
- Another way is Association; between source and destination; user specifies direction and cardinality.

### Association Directionality

- Uni-Directional.
- Bi-Directional.

### Association Cardinality

- Links maintained automatically
- Association Cardinality
  - ✓ One-to-one
  - ✓ One-to-many
  - ✓ Many-to-one
  - ✓ Many-to-many
- Object Copying and Versioning
  - ✓ Reference deleted, moved or copied



### Propagating Operations

- A useful property to form composite objects.
  - ✓ Delete Propagation.
  - ✓ Lock Propagation.
- Association Storage Properties.
  - ✓ Non-inline Associations.
  - ✓ Inline Associations.

### Lecture -42

**Association Storage Properties**

- Inline: association specific single attribute or array.
- Non-inline: system default association array, for all

**Space Requirements**

| Type of Association                          | Bytes per Link | Bytes per Allocated Association Array <sup>a</sup> |                | Space Required for <i>N</i> Links                    |
|----------------------------------------------|----------------|----------------------------------------------------|----------------|------------------------------------------------------|
|                                              |                | Embedded Reference to the Array <sup>b</sup>       | Array Overhead |                                                      |
| non-inline associations<br>(any cardinality) | 12             | 4 or 8                                             | 14             | $(12 \times N) + 4 + 14$ or $(12 \times N) + 8 + 14$ |
| inline association                           |                |                                                    |                |                                                      |
| to-one                                       | 8              | —                                                  | —              | 8 ( $N = 1$ )                                        |
| to-many                                      | 8              | 4 or 8                                             | 14             | $(8 \times N) + 4 + 14$ or $(8 \times N) + 8 + 14$   |
| short inline association                     |                |                                                    |                |                                                      |
| to-one                                       | 4              | —                                                  | —              | 4 ( $N = 1$ )                                        |
| to-many                                      | 4              | 4 or 8                                             | 14             | $(4 \times N) + 4 + 14$ or $(4 \times N) + 8 + 14$   |

**Uni-Directional Associations**

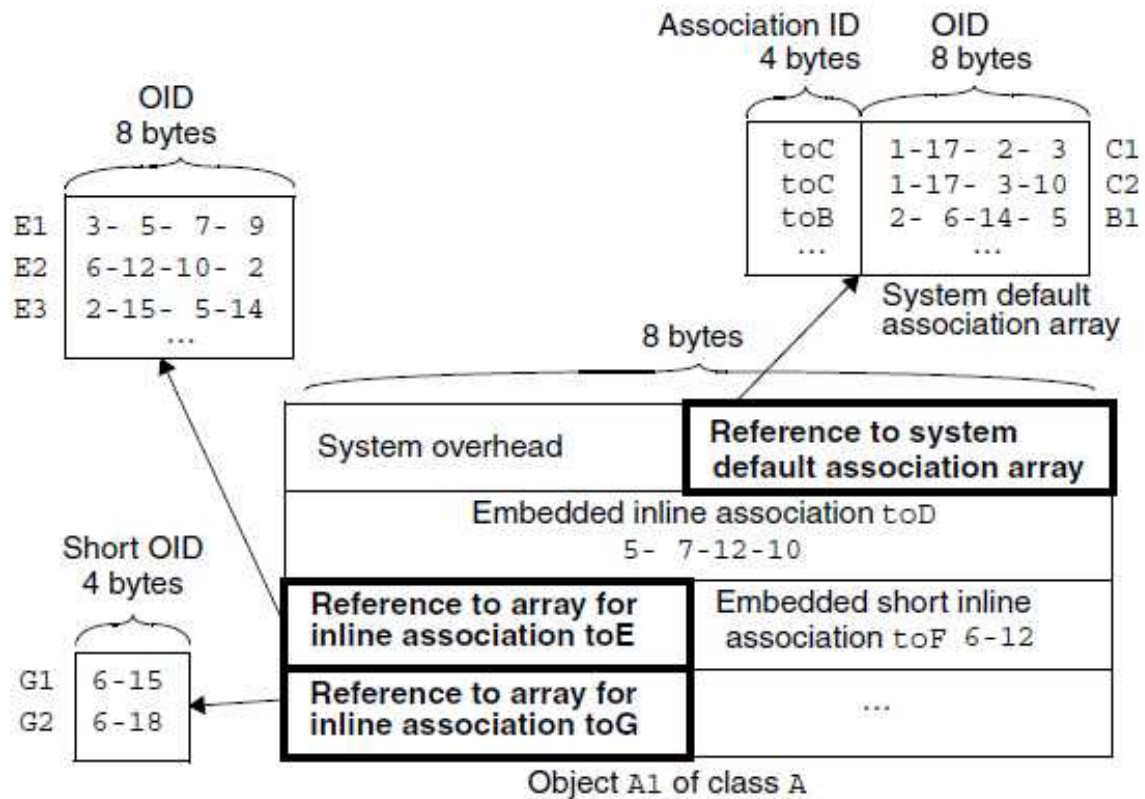
- To-one:       ooRef(className) linkName : bSpec {, bSpec};
- To-many:      ooRef(className) linkName[ ] : bSpec {, bSpec};
- class Salesperson: public ooObj {  
    public:  
    ...  
    ooRef(Contact) contacts[ ] : copy(delete); //Association  
    // ooRef(Contact) contacts[ ]; Attribute  
    //  
};
- class Contact: public ooObj {  
    public:  
    ...  
};

**Bi-directional Associations**

- One-to-one: `ooRef(className) linkName <-> inverseLinkName [: bSpec {, bSpec}];`
- One-to-many: `ooRef(className) linkName[ ] <-> inverseLinkName [: bSpec {, bSpec}];`
- Many-to-one: `ooRef(className) linkName <-> inverseLinkName[ ][: bSpec {, bSpec}];`
- Many-to-many: `ooRef(className) linkName[ ] <-> inverseLinkName[ ][: bSpec {, bSpec}];`

#### Inline Association

- `// DDL file`
- `class Salesperson: public ooObj {  
    public:  
    ...  
    inline ooShortRef(Contact) contacts[ ] <-> salesRep;  
};`
- `class Contact: public ooObj {  
    public:  
    ...  
    inline ooShortRef(SalesPerson) salesRep <-> contacts[ ];  
};`
- `class A: public ooObj {public: // Non-inline one-to-one association toB  
    ooRef(B) toB <-> toA;  
    // Non-inline one-to-many association toC  
    ooRef(C) toC[ ] <-> toA;  
    // Inline one-to-one association toD  
    inline ooRef(D) toD <-> toA;  
    // Inline one-to-many association toE  
    inline ooRef(E) toE[ ] <-> toA;  
    // Short inline one-to-one association toF  
    inline ooShortRef(F) toF <-> toA;  
    // Short inline one-to-many association toG  
    inline ooShortRef(G) toG[ ] <-> toA;  
};`



### Inline & Non-inline Association

- Non-inline
  - ✓ Memory efficient
  - ✓ An implicit limit on references for a single object
  - ✓ Referencing inefficient
- Inline
  - ✓ Memory inefficient
  - ✓ Access efficient
  - ✓ More objects can be referenced

### Storage Type

- Storage type can be changed
  - ✓ Change in DDL
  - ✓ Re-compile ddl file and re-building application with new file

### Behavior Specification

- Behavior specification consists of:
  - ✓ Propagation
  - ✓ Copying
  - ✓ Versioning

**Requesting Propagation Operations**

- Delete Propagation
- Lock Propagation

**Object Copying Behavior**

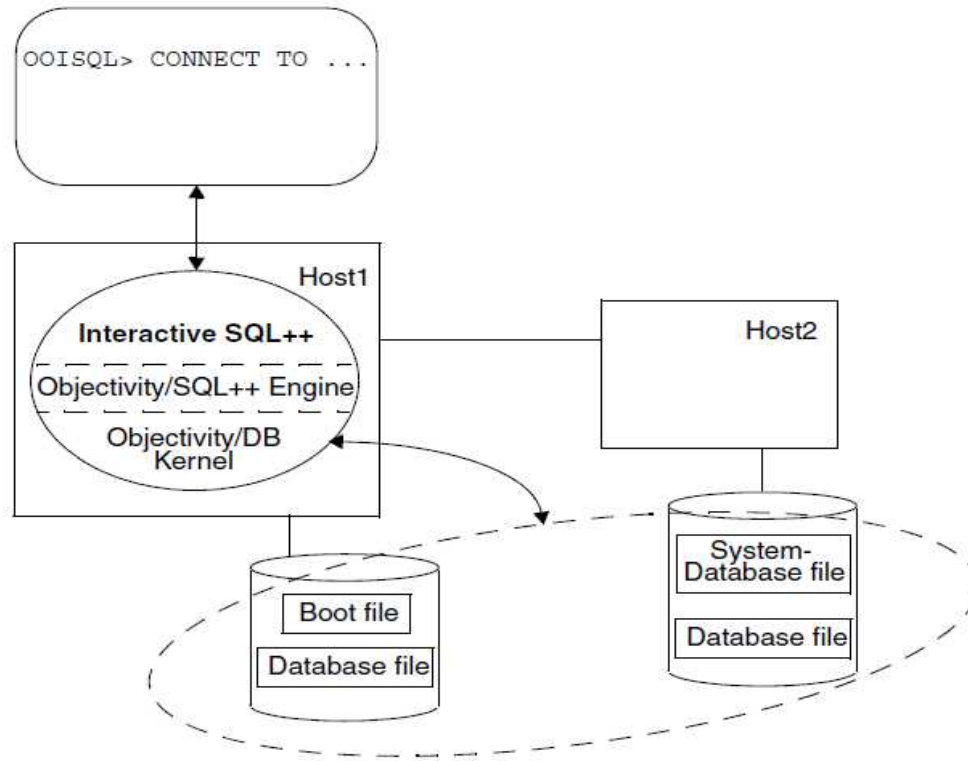
- traversalPath : copy(copy);
- traversalPath : copy(move);
- traversalPath : copy(delete);

**Object Versioning Behavior**

- traversalPath : version(copy);
- traversalPath : version(move);
- traversalPath : version(delete);
- class Salesperson: public ooObj {
  - public:
  - ...
  - ooRef(Contact) contacts[ ] <-> salesRep : lock(propagate), copy (move), version(move);
- };
- class Contact: public ooObj {
  - public:
  - ...
  - ooRef(Salesperson) salesRep <-> contacts[ ] : copy(copy);
- };

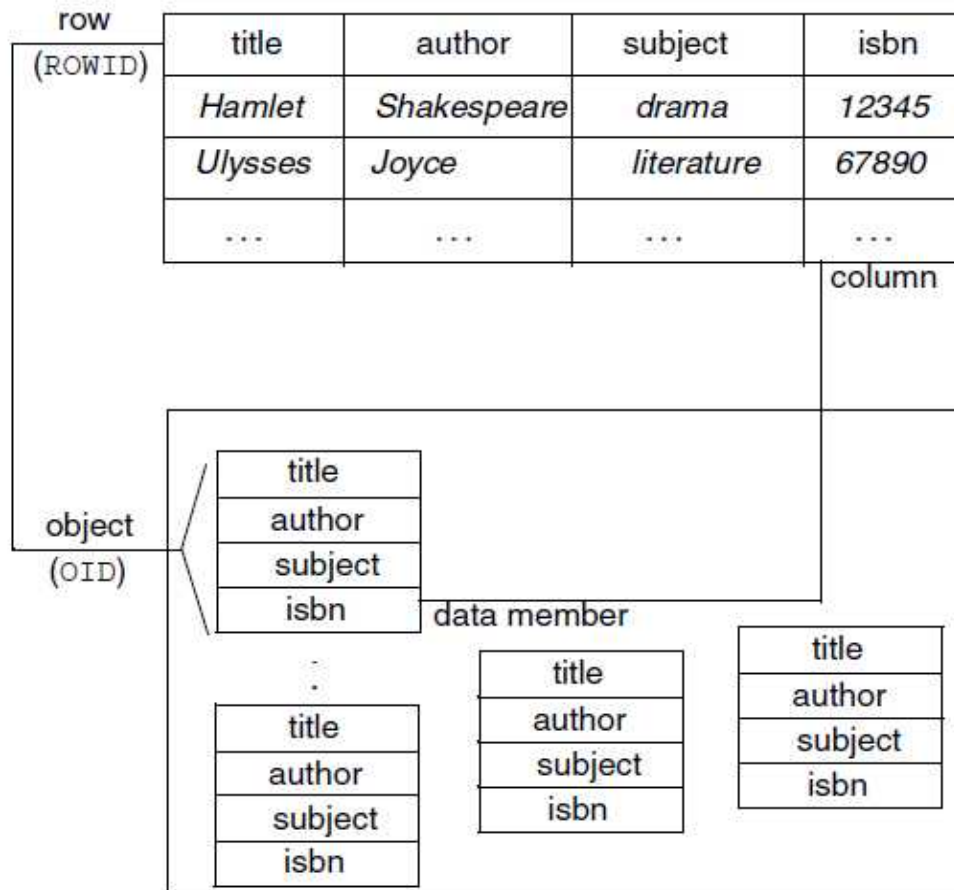
**Objectivity/SQL++**

- Operates in two modes
  - ✓ Interactive
  - ✓ Through ODBC
- Both use O/SQL++ engine to transform operations and handover to O/DB kernel



### Database Concepts and Structures

- Table: The objects of a particular Objectivity/DB class
- Column: Includes field types beyond those supported by SQL.
- Row: An object
- Row Id: OID
- Database: Objectivity/DB FDB
- Join: Objectivity/DB associations
- Index: Objectivity/DB index



- ```

class Book : public ooObj {
public:
    char _title[256];
    ooRef(Loan) activeLoan <-> activeBook;
};

```
- ```

class Loan : public ooObj {
public:
 ooSQLTimestamp _dueDate;
 ooRef(Book) activeBook <-> activeLoan;
 ooRef(Patron) patron <-> loans[];
};

```
- ```

class Patron : public ooObj {
public:
    ooVString _firstName; ooVString _lastName;
    ooRef(Loan) loans[ ] <-> patron;
};

```

- `SELECT patron._lastName, book._title, loan._dueDate FROM loan, patron, book
WHERE loan.patron = patron.oid AND loan.activeBook = book.oid;`
- Likewise we can also access inherited attributes
- We can define views.

Objectivity/SQL++

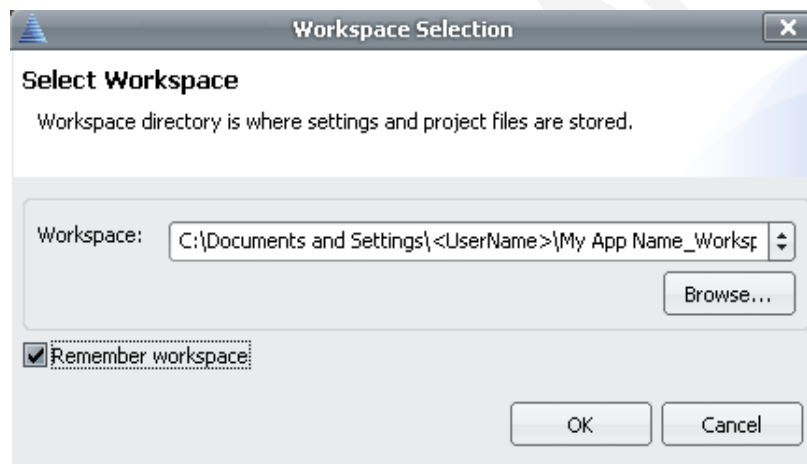
- Individual member of Array can be accessed (not for string)
- `class dept: public ooObj {
 public:
 struct { int field; char name[16]; }
 projs[4];
 ooVArray(char) string;
};`
- `SELECT <projs<2>.field> FROM dept`
- `WHERE <projs<2>.name> = 'SQL';`

Lecture -43**Using Objectivity****Using Objectivity (Assist)**

-

Installation Requirements

- Install jdk1.4 or higher.
- Download O/DB setup file and run 'Objectivity 10.1'.
- Place license in the root folder of O/DB.
- Install Visual Studio 2008.
- Install Objectivity DB 10.1.



- See lecture slides

Lecture -44**OO Distributed Database Management**

- Reference: M. Tamer Özsu, Patrick Valduriez “Principles of Distributed Database Systems, Third Edition (Chapter 15)

Distributed Database System

- Ingredients of Distributed DB System:
- Data Management on Multiple Sites.
- Same global/logical perspective for all DBs.
- Network connectivity.

Object Distribution Design

- Partitioning or Fragmentation of Database:
- Horizontal
- Vertical
- Hybrid (Combination of Horizontal and Vertical)

Horizontal Fragmentation

- Primary Horizontal Fragmentation (PHF)
- Derived Horizontal Fragmentation (DHF)
- DHF Issues: Arising from more specialized class: different classes may cause different partitioning
- If complex object, then containing object fragmentation may be difficult.
- Simplest PHF can be implemented through Abstract base class & derived classes.
- Once PHF has been defined for classes, DHF for superclasses defined using same predicates.
- Example
 - ✓ p1: horsepower < 200
 - ✓ p2: horsepower > 200

Vertical Fragmentation

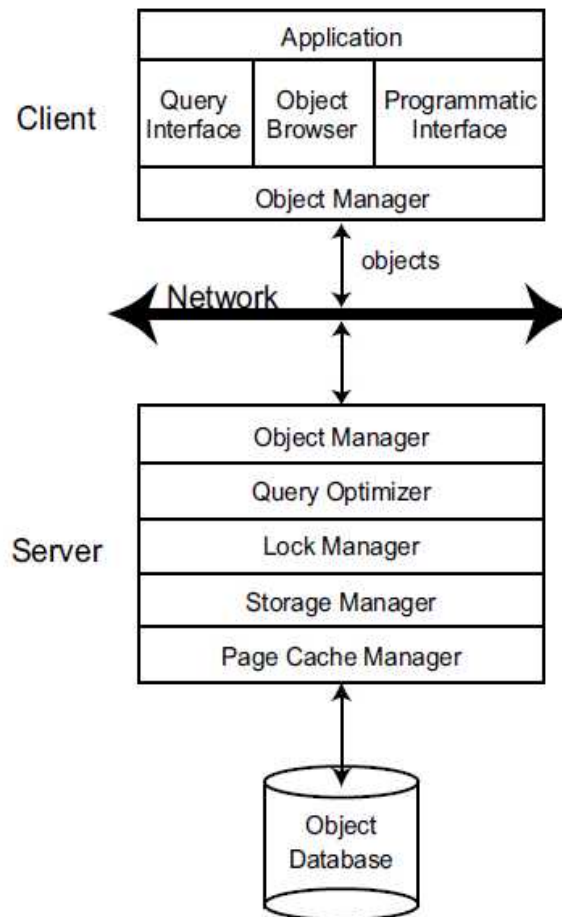
- Vertical fragmentation is considerably more complicated.
- Given a class C, fragmenting it vertically into $C_1, \dots, C_m$ produces a number of classes, each of which contains some of the attributes and some of the methods.

Architectural Issues

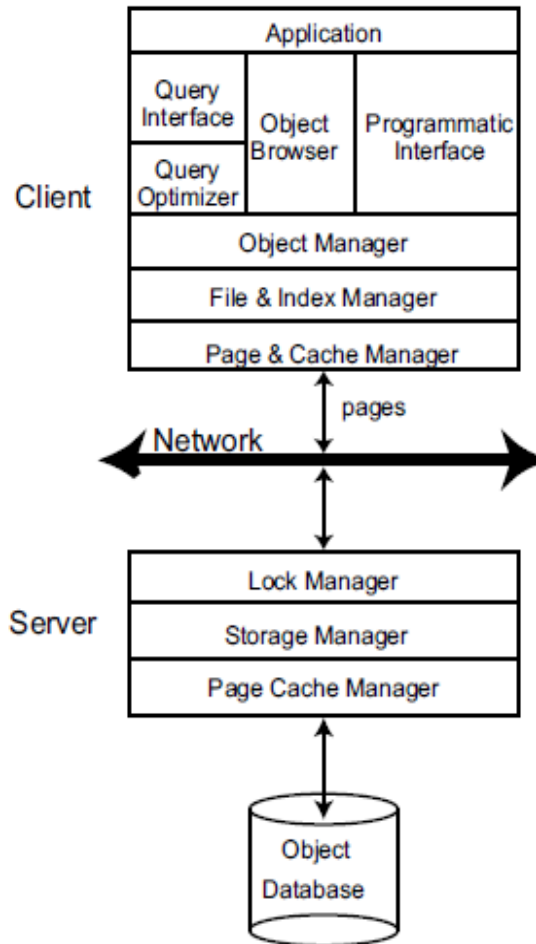
- Unit of Communication
 - ✓ Page
 - ✓ Object (s)
 - ✓ Page Server
- Design decision regarding the functions provided by clients and the server.
- In relational C/S systems, clients simply pass queries to server, which executes them and returns result tables to the client. This is referred to as function shipping.
- Since objects may be composite or complex, there may be possibilities for prefetching component objects when an object is requested.

Architectural C/S Alternatives

- Client Buffer Management
- Cache Consistency
- OID Management
- Pointer Swizzling
- Object Server
- Page Server

**Object Migration**

- One aspect of distributed systems is that objects move, from time to time, between sites.



Object State

- Ready
- Active
- Waiting
- Suspended

OID

- Implementing the identity of persistent objects generally differs from implementing transient objects, since only the former must provide global uniqueness. In particular, transient object identity can be implemented more efficiently.

Memory Management

-

Garbage Collection

- Garbage collection is a problem that arises in object databases due to reference-based sharing.
- Reference Counting
- Trace Back
 - ✓ Mark
 - ✓ Sweep
- Object Migration
- Distributed Object Storage

Lecture -45

Transaction Management Issues

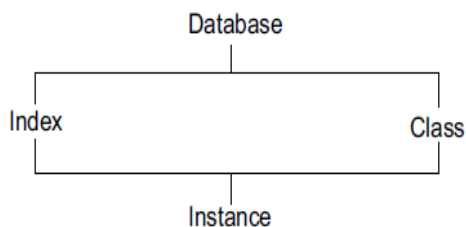
- Not studied much; raises certain issues
- Mostly server adopt page-based locking and flat transaction model;
- OODB applications need long transactions

Correctness Criteria

- Serializability: basic criterion
- Other alternatives exist
 1. Commutativity: two operations conflict if the results of different serial executions of these operations are not equivalent
 2. Syntactic Commutativity
- Other alternatives exist
 1. Commutativity:
 2. Semantic Commutativity: if semantics of operations is known then commutativity can be redefined, set operations like: Insert, Delete, Member
- Other alternatives exist
 1. Invalidation: whether or not the execution of one invalidates the other;
 - Considering Set example, Insert cannot be invalidated by any other operation, but a Member can be invalidated by a Delete if their arguments are same.
 2. Recoverability: an operation P is said to be recoverable with respect to operation Q if the value returned by P is independent of whether Q executed before P or not

Transaction Management

- Intra-object parallelism: storing attribute values in DB and methods as transactions; multiple invocations of methods can be invoked.
- Multi-granularity locking: different from Relational systems.



	S	X	IS	IX	SIX
S	+	-	+	-	-
X	-	-	-	-	-
IS	+	-	+	+	+
IX	-	-	+	+	-
SIX	-	-	+	-	-

Managing Class Lattice

- Inheritance causes problem, locking a class may require locking all subclasses
 1. Problematic when a class near to root is to be locked
- One option is to place IS/IX lock on subclasses; problematic too !

Ordered Sharing**Managing Containment**

- Composition and aggregation requires locking an object should lock all referred classes or all objects being referred.
- First effects concurrency, second one is time consuming.
- New locks defined.

	S	X	IS	IX	SIX	ISO	IXO	SIXO
S	+	-	+	-	-	+	-	-
X	-	-	-	-	-	-	-	-
IS	+	-	+	+	+	+	-	-
IX	-	-	+	+	-	-	-	-
SIX	-	-	+	-	-	-	-	-
ISO	+	-	+	+	-	+	+	+
IXO	-	-	-	-	-	+	+	-
SIXO	N	N	N	N	N	Y	N	N

Query Processing/Optimization

- Strength of Relational Query Processing and Optimization
 1. Has to operate on a simple structure, that is, a relation.
 2. Based on relational algebra.
 3. Properties of relational algebra operators.
- In contrast OODBMS has to:
 1. Operate on a complex objects.
 2. Execution cost involves database access paths and statistics; not available in case of OO systems.

OODBMS

- Pros:
 1. Support for complex objects: UDTs, Multimedia types; so suitable for some typical applications.
 2. Object modeled in its entirety, as a whole.
- Cons:
 1. Lack of standardization.
 2. Support of DBMSs and tools.
 3. Complexity of class hierarchy.

Overview of the Course

- Started with a discussion on Data Models.
- RDM being most dominant.
- Drawbacks of RDM.
- Push for new technology.
- Three approaches adopted.

Overview of the Course

- Three approaches
 1. Extended relational
 - Extensions in Data Model
 - Extensions in RDM Tools: Postgres, Starburst
 2. Object Relations
 - Built from scratch: Open illustra
 - Built on top of a Relational system: Open ODB/Adapter
 3. Pure OODBMS