The ODMG Standard

- ODMG (Object Database Management Group) 3.0 was released in 2000
  - As of 2013, ODMG 3.0 is the most recent release.
- Includes the data model (more or less)

- ODL: The object definition language
- OQL: The object query language
- A transaction specification mechanism
- Language bindings: How to access an ODMG database from C++, Smalltalk, and Java
Main Idea: *Host Language = Data Language*

- **Objects in the host language are mapped directly to database objects**

- Some objects in the host program are persistent. Think of them as “proxies” of the actual database objects. Changing such objects (through an assignment to an instance variable or with a method application) directly and transparently affects the corresponding database object.

- Accessing an object using its oid causes an “object fault” similar to page faults in operating systems. This transparently brings the object into the memory and the program works with it as if it were a regular object defined, for example, in the host Java program.
SQL Databases vs. ODMG

- **In SQL**: Host program accesses the database by sending SQL queries to it (using JDBC, ODBC, Embedded SQL, etc.)

- **In ODMG**: Host program works with database objects directly
  - For example, if Stud contains the oid of a persistent Student object, `Stud.Name = "John"` changes the student name to John both in the program and also in the database.
  - ODMG has the facility to send OQL queries to the database, but this is viewed as evil: brings back the impedance mismatch.
ODL: ODMG’s Object Definition Language

- Is rarely used, if at all!
  - *Relational databases*: SQL is the only way to describe data to the DB
  - *ODMG databases*: can do this directly in the host language
  - Why bother to develop ODL then?

- Problem: Making database objects created by applications written in different languages (C++, Java, Smalltalk) *interoperable*
  - Object modeling capabilities of C++, Java, Smalltalk are very different.
  - How can a Java application access database objects created with C++?

- *Hence*: Need a reference data model, a common target to which to map the language bindings of the different host languages
  - *ODMG says*: Applications in language A can access objects created by applications in language B if these objects map into a subset of ODL supported by language A

ODMG Data Model

- Classes + inheritance hierarchy + types
  - ODMG ODL is an extension of the IDL in CORBA

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

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

Example

```java
interface PersonInterface: Object { // Object is the ODMG topmost interface
    attribute String Name;
    attribute String SSN;
    Integer Age();
}

class PERSON: PersonInterface // inherits from ODMG interface
    // note: extents have names
    { extent PersonExt
        // note: extents have names
        keys SSN, (Name, PhoneN) : PERSISTENT;
        { attribute ADDRESS Address;
            attribute Set<String> PhoneN;
            attribute enum SexType {m,f} Sex;
            attribute date DateOfBirth;
            relationship PERSON Spouse; // note: relationship vs. attribute
            relationship Set<PERSON> Child;
            void add_phone_number (in String phone); // method signature
        }
    }

struct ADDRESS { // a literal type (for pure values)
    String StNumber;
    String StName;
}
```
More on the ODMG Data Model

- Can specify keys (also foreign keys – later)

- Class extents have their own names – this is what is used in queries
  - As if relation instances had their own names, distinct from the corresponding tables

- Distinguishes between relationships and attributes
  - Attribute values are literals
  - Relationship values are objects
  - ODMG relationships have little to do with relationships in the E-R model – do not confuse them!!

Example (contd.)

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

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

class STUDENT extends PERSON {  // inheritance from a class
    ( extent StudentExt )
    attribute Set<String> Major;
    relationship Set<COURSE> Enrolled;
}
class COURSE: Object ( // inheritance from an interface
    ( extent CourseExt )
    attribute Integer CrsCode;
    attribute String Department;
    relationship Set<STUDENT> Enrollment;
}

• Referential integrity: If JoePublic takes CS532, and CS532 \in JoePublic.Enrolled, then deleting the object for CS532 will delete it from the set JoePublic.Enrolled

• Still, the following is possible:
  CS532 \in JoePublic.Enrolled but JoePublic \notin CS532.Enrollment

• Question: Can the DBMS automatically maintain consistency between JoePublic.Enrolled and CS532.Enrollment?

Referential Integrity (Contd.)

Solution:

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

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

Example: Simple OQL Query

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

- Can hardly tell if this is OQL or SQL
- Note: Uses the name of the extent of class PERSON, not the name of the class
Example: A Query with Method Invocation

- Method in the SELECT clause:
  
  ```sql
  SELECT M.frameRange(100, 1000)
  FROM MOVIE M
  WHERE M.Name = “The Simpsons”
  ```

- Method with a side effect:
  
  ```sql
  SELECT S.add_phone_number(“555-1212”)
  FROM PersonExt S
  WHERE S.SSN = “123-45-6789”
  ```

- This feature blurs the boundary between the data manipulation and query languages

OQL Path Expressions

- Path expressions can be used with attributes:
  
  ```sql
  SELECT DISTINCT S.Address.StName
  FROM PersonExt S
  WHERE S.Name = “Smith”
  ```

- As well as with relationships:
  
  ```sql
  SELECT DISTINCT S.Spouse.Name
  FROM PersonExt S
  WHERE S.Name = “Smith”
  ```
Path Expressions (Contd.)

- Must be type consistent: the type of each prefix of a path expression must be consistent with the method/attribute/relationship that follows.

- For instance, is S is bound to a PERSON object, then S.Address.StName and S.Spouse.Name are type consistent:
  - PERSON objects have attribute Address and relationship Spouse
  - S.Address is a literal of type ADDRESS; it has an attribute StName
  - S.Spouse is an object of type PERSON; it has a attribute Name, which is inherited from PersonInterface

Path Expressions (Contd.)

- Is P.Child.Child.PhoneN type consistent (P is bound to a PERSON objects)?
  - In some query languages, but not in OQL!

- Issue: Is P.Child a single set-object or a set of objects?
  - If it is a set of PERSON objects, we can apply Child to each such object and P.Child.Child makes sense (as a set of grandchild PERSON objects)
  - If it is a single set-object of type Set<Person>, then P.Child.Child does not make sense, because such objects do not have the Child relationship

- OQL uses the second option. Can we still get the phone numbers of grandchildren? – Must flatten out the sets:
  - flatten(flatten(P.Child).Child).Phone
Nested Queries

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

What does the next query do?

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

Aggregation and Grouping

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

```sql
SELECT name: S.Name,
      count: count ( SELECT E.CrsCode
                     FROM S.Enrolled E
                     WHERE E.Department = "CS" )
FROM StudentExt S
```
Aggregation and Grouping (Contd.)

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

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

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

GROUP BY as an Optimizer Hint

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

The query optimizer would compute the inner query for each s∈StudentExt, so S.Enrolled will be computed for each s.
If enrollment information is stored separately (not as part of the STUDENT Object), then given s, index is likely to be used to find the corresponding courses.
Can be expensive, if the index is not clustered

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

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

- A set of interfaces and class definitions that allow host programs to:
  - Map host language classes to database classes in ODL
  - Access objects in those database classes by direct manipulation of the mapped host language objects

- Querying
  - Some querying can be done by simply applying the methods supplied with the database classes
  - A more powerful method is to send OQL queries to the database using a statement-level interface (which makes impedance mismatch)

Java Bindings: Simple Example

```java
public class STUDENT extends PERSON {  // Java class definition
    public DSet Major;
    ............
}
```

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

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

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

---

Cannot say “set of strings” – a Java limitation
Language Bindings: Issues

- Host as a data manipulation language is a powerful idea, but:
  - Some ODMG/ODL facilities do not exist in some or all host languages
  - The result is the lack of syntactic and conceptual unity

- Some issues:
  - Specification of persistence (which objects persist, i.e., are automatically stored in the database by the DBMS, and which are transient)
    - First, a class must be declared persistence capable (differently in different languages)
    - Second, to actually make an object of a persistence capable class persistent, different facilities are used:
      - In C++, a special form of `new()` is used
      - In Java, the method `makePersistent()` (defined in ODMG-Java interface `Database`) is used
  - Representation of relationships
    - Java binding does not support them; C++ and Smalltalk bindings do
  - Representation of literals
    - Java & Smalltalk bindings do not support them; C++ does

Java Bindings: Extended Example

- The `OQLQuery` class:
  ```java
class OQLQuery {
    public OQLQuery(String query);  // the query object constructor
    public bind(Object parameter);  // explained later
    public Object execute();        // executes queries
    ...  ...  several more methods  ...  ...
}
```

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

- Courses taken exclusively by CS students in Spring 2002:

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

Interface DCollection

- Allows queries (select) from collections of database objects

- DSet inherits from DCollection, so, for example, the methods of DCollection can be applied to variables courses, students (previous slide)

```java
public interface DCollection extends java.util.Collection {
    public DCollection query(String condition); // query constructor
    public Object selectElement(String condition); // get some element
    public Boolean existsElement(String condition); // test non-emptiness
    public java.util.Iterator select(String condition); // get collection & iterator
}
```
Extended Example (Cont.)

- **query(condition)** – selects a subcollection of objects that satisfy condition:
  
  ```
  DSet seminars;
  seminars = (DSet) courses.query("this.Credits = 1");
  ```

- **select(condition)** – like query( ), but creates an iterator; can now scan the selected subcollection object-by-object:
  
  ```
  java.util.Iterator seminarIter;
  Course seminar;
  seminarIter = (java.util.Iterator) courses.select("this.Credits=1");
  while ( seminar=seminarIter.next( ) ) {
      ...
      ...
  }
  ```

---

**CORBA:**

**Common Object Request Broker Architecture**

- Distributed environment for clients to access objects on various servers

- Provides *location transparency* for distributed computational resources

- Analogous to *remote procedure call* (RPC) and *remote method invocation* in Java (RMI) in that all three can invoke remote code.

- But CORBA is more general and defines many more protocols (e.g., for object persistence, querying, etc.)
Interface Description Language (IDL)

- **Specifies interfaces only** (i.e., classes without extents, attributes, etc.)

- **No constraints or collection types**

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

CORBA Architecture
Object Request Broker (ORB)

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

ORB Server Side

- **Library.idl** → **IDL Compiler** → Library-stubs.c, Library-skeleton.c → Method signatures to interface repository

- **Server skeleton**: Library-skeleton.c
  - Requests come to the server in OS/language/machine independent way
  - Server objects are implemented in some concrete language, deployed on a concrete OS and machine
  - Server skeleton maps OS/language/machine independent requests to calls understood by the concrete implementation of the objects on the server

- **Object adaptor**: How does ORB know which server can handle which method calls? – Object adaptor, a part of ORB
  - When a server starts, it registers itself with the ORB object adaptor
  - Tells which method calls in which interfaces it can handle. (Recall that method signature for all interfaces are recorded in the interface repository).

- **Implementation repository**: remembers which server implements which methods/interfaces (the object adaptor stores this info when a server registers)
ORB Client Side

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

Static Invocation

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

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

Dynamic Invocation (Contd.)

- Example: IDL module with different search capabilities
  module Library {
    interface library1 {
      string searchByKeywords(in string keywords);
      string searchByAuthorTitle(in string author, in string title);
    }
    interface library2 {
      void searchByTitle(in string title, out string result);
      void searchByWildcard(in string wildcard, out string result);
    }
  }

Instead of using the stub, the client application:
- Examines the fields in the form filled out by the user
- Examine the interface repository using dynamic invocation API provided by CORBA
- Decides which methods it can call with which arguments
- Constructs the actual call – next slide
Dynamic Invocation API

- Provides methods to query the interface repository
- Provides methods to construct machine-independent requests to be passed along to the server by the ORB
- Once the application knows which method/interface to call with which arguments, it constructs a request, which includes:
  - Object reference (which object to invoke)
  - Operation name (which method in which interface to call)
  - Argument descriptors (argument names, types, values)
  - Exception handling info
  - Additional “context” info, which is not part of the method arguments

Note: The client stub is essentially a piece of code that uses the dynamic invocation API to create the above requests. Thus:
- With static invocation, the request is created automatically by the IDL compiler
- With dynamic invocation, the programmer has to manually write the code to create and invoke the requests, because the requisite information is not available at compile time

Interoperability within CORBA

- ORB allows objects to talk to each other if they are registered with that ORB; can objects registered with different ORBs talk to each other?

- General inter-ORB protocol (GIOP): message format for requesting services from objects that live under the control of a different ORB
  - Often implemented using TCP/IP
  - Internet inter-ORB protocol (IIOP) specifies how GIOP messages are encoded for delivery via TCP/IP
Inter-ORB Architecture

![Diagram of Inter-ORB Architecture]

CORBA Services

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

- PSS – a standard way for data stores (e.g., databases, file systems) to define interfaces that can be used by CORBA clients to manipulate the objects in that data store

  - On the server:
    - Objects are in storage homes (e.g., classes)
    - Storage homes are grouped in data stores (e.g., databases)

  - On the client:
    - Persistent objects (from the data store) are represented using storage object proxies
    - Storage object proxies are organized into storage home proxies

- Clients manipulate storage object proxies directly, like ODMG applications do
Naming Services

- Enables programs to refer to (persistent) objects
- Represented as a directed acyclic graph
  - Object name is the concatenation of the labels in a path
  - e.g., /Library/myTownLibrary/search

- Server may implement CORBA interface NamingContext
  - Void bind (in Name n, in Object o)
  - Object resolve (in Name n)

Object Query Services (OQS)

- OQS makes it possible to query persistent CORBA objects
- Supports SQL and OQL

- Does two things:
  - Query evaluator: Takes a query (from the client) and translates it into a query that is appropriate for the data store at hand (e.g., a file system does not support SQL, so the query evaluator might have quite some work to do)
  - Query collection service: Processes the query result.
    - Creates an object of type collection, which contain references to the objects in the query result
    - Provides an iterator object to let the application to process each object in the result one by one
Transaction and Concurrency Services

- **Transactional services:**
  - Allow threads to become transactions. Provide
    - `begin()`
    - `rollback()`
    - `commit()`
  - Implement *two-phase commit protocol* to ensure atomicity of distributed transactions

- **Concurrency control services:**
  - Allow transactional threads to request and release locks
  - Implement *two-phase locking*
  - Only supports — does not enforce — isolation. Other, non-transactional CORBA applications can violate serializability