Part V

Object-oriented Data Models

Overview

- Object-oriented Database Management Systems
- Concepts of OO Data Models
- Concepts of ODBMS
- The ODMG Standard
Object-oriented Database Management Systems

Based on the development of
- Object-oriented analysis and design (OOA, OOD)
- Object-oriented programming (OOP)
- Object-oriented distributed computing (e.g. CORBA)

Objectives:
- More and advanced semantic concepts to describe real world facts
- More efficiency for non-standard applications
- Overcome impedance mismatch between RM and OOP

Definition

An **Object-oriented Database Management System (ODBMS)** is a database management systems that implements an object-oriented data model.

Problem: there is no ONE object-oriented model, but several offering similar concepts

Basic requirements were listed in *The Object-oriented Database System Manifesto* (Atkinson et al., 1989)

ODMG supposed to provide a standard for ODBMS (1993), but limited support by commercial systems
Advanced Database Models

Object-oriented Data Models

Object-oriented Database Management Systems

History

- First research prototypes in the early '80s (ORION, ITASCA, EXODUS)
- First commercial systems in the late '80s/ early '90s (GemStone, ONTOS, Objectivity, ObjectStore, Versant, Poet, O²)
- First version of the ODMG standard in 1994
- Heydays during the mid-'90s
- Systems easily integrated support for Java, the WWW, XML, multi-media, spatial data, etc.
- Since mid-'90s: RDBMS integrated object-oriented concepts into their systems → ORDBMS
- Few commercial ODBMS survived until today

Different Approaches

Based on OOP data models: these ODBMS used the standardized data models of popular OO programming languages such as C++ (ONTOS, Objectivity, ObjectStore, Versant, Poet) or Smalltalk (GemStone), later on Java (Jasmine, JD4O), and added DBMS functionality (persistence, TXNs, collection types, queries, ...)

Extensions of relational models: introduce object-oriented concepts (types/classes, inheritance, object identity, methods, ...) in a relational model (Postgres, Illustra) or build on top of existing DBMS (Oracle, DB2, ...) → object-relational DBMS

Innate OO database models: developed independently of existing models and systems (O², ORION, Itasca)
Object Database Management Systems

- Most successful system were tightly integrated with OOPL
  - C++: Versant, Objectivity, Poet (now Versant Fast Objects)
  - Smalltalk: GemStone
  → solved problem of Impedance Mismatch!
- Object-relational systems provide similar concepts but are decoupled from programming language (advantages of logical data independence)

Advantages of ODBMS

- Provide semantically rich object-oriented modeling constructs
- Solve Impedance Mismatch
- Decreased implementation efforts
  - Developer needs knowledge of only one data model
  - No mapping code/layer required
- Efficiency for applications with complex data (navigational access, no joins required)
Disadvantages of ODBMS

- No logical data independence
  - Application changes may trigger schema changes
  - Schema changes require application changes
- Existing standard (ODMG) is hardly implemented
- Lack of interoperability
- Many systems with insufficient support for typical DBMS functionality
  - Logical views
  - Query languages
  - Query/access optimization
  - TXNs
  - Distribution
  - ...

The Object-oriented Database System Manifesto

- Though there is no common OO data(base) model, all models share important characteristics
- *The Object-oriented Database System Manifesto* summarized minimum requirements
  - Mandatory features: the Golden Rules
    - Object-oriented concepts
    - Database concepts
  - Optional features: the goodies
  - Open choices
### OO Manifesto: Golden Rules /1

- **OO concepts (part I of mandatory concepts)**
  - Complex objects
  - Object identity
  - Encapsulation
  - Types and/or Classes
  - Class and/or Type Hierarchies
  - Overriding, overloading and late binding
  - Computational completeness
  - Extensibility

### OO Manifesto: Golden Rules /2

- **DBMS concepts (part II of mandatory concepts)**
  - Persistence
  - Secondary storage management
  - Concurrency
  - Recovery
  - Ad Hoc Query Facility
Optional concepts that may be implemented
- Multiple inheritance
- Type checking and type inferencing
- Distribution
- Design transactions
- Versions

A very short discussion of further design considerations
- Programming paradigm
- Representation system
- Type system
- Uniformity
Complex Objects and Extensibility

- System must provide basic types: integer, float, boolean, character, strings
- System must provide at least the following type constructors for complex objects
  - tuple
  - set
  - list
- Complex objects may require specific operations, e.g.
  - deep or shallow delete
  - deep or shallow copy
- Extensibility: user-defined types can be used in the same way as system-defined types

Object Identity

The **object identity** of an object represents the fact, that it has an existence which is independent of all its values. Two objects are **identical** if they have the same object identity. Two objects are **equal** if they have the same values.

- Used for referencing objects (similar to OOPL pointers or references)
- Can be mapped to physical location or logical address (e.g. table, primary key)
- May be system- or user-generated
Object Identity /2

- Equality for complex objects
  - objects are **shallow equal** if they refer to identical component objects
  - objects are **deep equal** if they refer to (deep) equal component objects

- Object identity must be unique and non-volatile
  - in time: can not change and can not be re-assigned to another object, when the original object is deleted
  - in space: must be unique within and across database boundaries

Types and/or Classes

- Manifesto requires support of types or classes
- Abstraction of common features of a set of objects with the same characteristics
- Differences in definitions exist for OO programing languages

A **type** describes the **intension**, i.e. the internal (the set of encapsulated, possibly complex attributes) and external structure (the interface of the type with methods and their signatures).

A **class** consists of a type description, and in addition includes the notions of an object factory (e.g. constructors) and of a class **extension** which represents the set of all instances of each particular class.
Encapsulation

**Encapsulation** represents the postulation, that only the interface part of a type is visible to the users of the type, the implementation of the object is seen only by the type designer.

- Compile-time aspect of OOPL
  - Access modifiers, e.g. `public` (accessible for all), `private` (accessible within class), `protected` (accessible within class or derived class) checked by compiler
  - Strict encapsulation allows access to internal structure (attributes) only via interface (methods)
- Becomes run-time aspect for DBMS

Type and Class Hierarchies

- Type and class hierarchies imply several aspects

**Specialization inheritance**, or intensional specialization, between a super-type `A` and a subtype `B` means that the set of attributes and methods defined in `A` are a subset of the attributes and methods defined in `B`.

**Substitution inheritance**, or extensional specialization, between a superclass `A` and a subclass `B` means that the set of instances of `B` are a subset of the instance of class `B`. **Polymorphism** means that all instances of class `B` can be used wherever objects of class `A` can be used (substitutability).
Advanced Database Models
Object-oriented Data Models
Concepts of OO Data Models

Type and Class Hierarchies /2

Advanced Database Models
Object-oriented Data Models
Concepts of OO Data Models

Overriding, Overloading and Late Binding

Different aspects of polymorphism:

**Method overloading:** allows method signatures to be re-defined within a class or in derived classes with different return or parameter types.

**Method overriding:** allows method implementations to be re-defined in derived classes.

**Late Binding:** the correct implementation of an overridden method of an object is determined and executed during run-time.
Computational Completeness

- ODBMS require tight integration with computationally complete programming language for
  - Method implementations
  - Navigational access
  - Control of DBMS functionality (persistence, TXNs, etc.)
- Not typical for RDBMS: only offer relational completeness
- Computational completeness often realized by means of integration with language of implemented data model (C++, Smalltalk, Java, ...)
- \( O^2 \) and object-relational systems also offer own languages

Concepts of ODBMS

- ODBMS require non-standard functionality beyond definition of data model
  - How are schemas defined (DDL) and implemented?
  - How is data made persistent and modified (DML)?
  - How is data accessed and queried?
  - How is consistency and concurrency controlled (advanced TXN models)
Schema Definition and Implementation

**Schema definition:**

**Application-independent:** ODMG ODL files

**Application-dependent:** source or object files of C++, Java, Smalltalk, C#, etc. application

**Schema implementation:**

**Pre-processor:** tool that takes source files (.h, .java, .odl, etc.) as input and creates modified or new (.odl) source code, possibly with added functionality / interfaces / super-classes, and, furthermore, installs the schema in the database

**Post-processor:** tool that derives the according schema information from an object file (.o, .obj, .class, etc.), possibly modifies the object file, and installs the schema

Sample Input Files

**Java:**

```java
public abstract class Person {
    private String firstname, lastname;
}
```

**C++:**

```cpp
class Person {
    private:
        char* _firstname, _lastname;
};
```

**ODL:**

```odl
class Person {
    extent persons) {
        attribute string firstname;
        attribute string lastname;
    }
```
Advanced Database Models
- Object-oriented Data Models
- Concepts of ODBMS

ODMG C++ Preprocessor

- ODL File
- ODL Preprocessor
- Generated C++ .h File
- C++ Compiler and Linker
- Executable DBMS Application

Generic Java Post-processor

- Database Classes
- Database Schema
- javac
- Database
- Other Classes
- Postprocessor
- modified .class File
- other .class Files
- Executable DBMS Application
Persistent-capable classes are created by schema definition and implementation. They can have persistent as well as transient instances.

Persistence independence requires that persistent and transient objects of a persistent-capable class can be handled uniformly.

Persistence orthogonality requires that persistence is independent of the application schema types/classes, e.g. that no common base class or interface must be inherited or implemented.

Persistent objects can be modified by accessing their attributes and methods according by means of the OO programming language (restricted by encapsulation).

Persistent objects can be created in several ways:
- Explicit persistence
- Named root objects
- Persistence by reachability (transitive persistence)

Persistent objects can be deleted by either:
- Explicit removal from database → may lead to dangling references and memory leaks
- Database garbage collection for unreferenced objects

Alternatively, data manipulations can be carried out using the query language of some systems.
Explicit Persistence

- Database run-time system offers explicit functionality to create a persistent object or make existing object persistent
- Create a persistent object, e.g. ODMG C++ Binding with overloaded `new`-operator

```c++
d_Db* db;
d_Ref<Person> p1 = new(db,"Person") Person("Smith","Carl");
```

- Make a transient object persistent, e.g. ODMG Java Binding

```java
Person p1 = new Person("Smith","Carl");
db.makePersistent(p1);
...
db.deletePersistent(p1);
```

Named Objects

- Retrieving the first object from an ODB by unique names
- Other objects can be accessed starting from this root object by following references
- Typical root objects: tree roots and collections
- Example: ODMG Java Binding

```java
City c1 = new City("NY", 45.67, 12.34);
db.bind(c1, "NY");
...
City c = (City) db.lookup("NY");
```
Advanced Database Models
Object-oriented Data Models
Concepts of ODBMS

Named Objects /2

Root Objects:
"NY"
"Paris"

Persistence By Reachability

- Possible consistency problem: object $a$ is made persistent and referenced objects $b$ and $c$ remain transient
  - References from persistent to transient storage
  - References become invalid when application is stopped or $b$ and $c$ are removed from memory
- Solution: if one object is made persistent, all transitively reachable objects are made persistent, too
- Example: ODMG Java Binding, JDO
Persistence By Reachability /1

1) Transient Objects

Primary Storage
Main Memory

Secondary Storage
Database

2) makePersistent

3) Program Exit

Persistence By Reachability /2

1) Transient Objects

Primary Storage
Main Memory

Secondary Storage
Database

2) makePersistent
   a) object itself
      a

   b) all reachable Objects

   a

   b

   c
OO Query Languages

- What are the results of OO queries?
  - Relations of tuples (without defined object type)
  - Existing objects
  - Objects created by queries

- How are queries used?
  - Ad-hoc queries: arbitrary queries, e.g. from command line tools
  - Query API: usage from within application program

- How are OO features accessed by query language?
  - Complex objects and navigation
  - Type hierarchies
  - Methods

OO Query Languages: Results

- **Relational query semantics:** returns set, bag, or list of struct (tuple) of literal data types as in relational query languages

- **Object selection semantics:** selects existing persistent objects from the database

- **Object creation semantics:** creates new (transient) objects based on values derived from persistent objects by calling a constructor in the `SELECT` clause ($\pi$)

- ODMG OQL supports all result types

- Most commercial systems support only object selection
OO Query Languages: Usage /1

- Ad-hoc queries, e.g. using ODMG interface OQLQuery, require input
  - Named root objects
  - Named extensions
  - Collections bound to variable

```java
OQLQuery query = new OQLQuery();
query.create("students");
DBag allStudents = (DBag) query.execute();
```

OO Query Languages: Usage /2

- Alternative: query interface bound to collection classes for object selection
  - E.g. `query()` method defined in ODMG collection classes
  - Specification of predicate

```java
DBag badStudents = (DBag)
student.query("grade > C");
DBag myStudents = (DBag)
student.query("EXISTS s IN this.lectures:
lecture.title = 'ADBM'");
```
OO Query Languages: OO Features

- Complex objects and navigation
  - Following path expressions using "."-notation
  - Collection-type references require unnesting or nested queries
- Type inheritance / specialization
  - Access to flat or deep extension
- Methods
  - Overriding, Overloading, Late Binding must be supported
  - Problem: may have side effects, e.g. read-only query calls method that changes attribute values

Extended Transactional Concepts

- Support for ACID transactions in most systems
- Also extended transactional concepts
  - Nested transactions: hierarchical structure of transactions to support complex operations on complex objects
  - Design transactions: typical requirement in non-standard applications long transactions avoiding concurrency problems, e.g. including check-out/check-in, possibly in conjunction with workspaces and versioning
The ODMG Standard

- Not well supported, but summarizes common features of ODBMS
- Influenced development of, for instance, JDO mapping standard
- Consists of several pieces
  - ODMG Object Model
  - Object Definition Language (ODL)
  - Object Query Language (ODL)
  - Language Bindings
    - C++
    - Smalltalk
    - Java

Object Definition language ODL

- Based on OMG IDL (CORBA Interface Definition Language)
- Implementation-independent schema definition
- Provides means to describe
  - Modules (application schemata)
  - Interfaces and Classes
  - Inheritance
  - Attributes and Relationships
  - Operations (method declarations)
  - Exceptions
- Translated to implementation language using pre-processor
class Employee {
    extent employees;
    attribute long employeeNr;
    attribute struct Name {
        string firstname;
        string lastname } name;
    attribute Date dob;
    attribute List<string> tel;
    ...
    void raise_salary (in short amount);
};

class Student : Person {
    extent students, key matrnr;
    attribute char matrnr[6];
    attribute string faculty;
    attribute set<struct<float grade, string lecture>> grades;

    relationship Person mother inverse Person::child;
    relationship Person father inverse Person::child;

    float avgGrade () raises (no_Grade);
    void enlist (in string faculty) raises (already_enlisted);
};
ODMG ODL: Inheritance

- Multiple inheritance only for interfaces

```java
interface Student {...};
interface Employee {...};
interface PhDStudent : Student, Employee {...};
```

- Class inheritance with `extends` supports only singular inheritances

```java
class Book extends Publication {...};
```
Object Query Language (OQL)

- Declarative object-oriented query language based on O^2 system
- Very comprehensive and flexible, but only small subset supported by most systems

Syntax:
- SFW-style queries
- Additionally: every named object (e.g. root) or collection (e.g. class extent) is a query
- Specification of predicates for collection-bound queries

ODMG OQL: Simple Example Queries

```sql
SELECT e FROM employees e;
employees;
newyork.neighbor_cities;
```
ODMG OQL: SFW Queries /1

- **SELECT-clause**
  - Allows method calls and sub-queries
  - Allows constructor calls
  - **DISTINCT** for result type `set<...>`

```sql
SELECT DISTINCT STRUCT (e.name, projects:
  SELECT p.projectID
  FROM e.participates_in p)
FROM employee e;
```

Result type:
`set<struct<name: string,
  projects: bag<long>>>`

ODMG OQL: SFW Queries /2

- Single object as result type using **ELEMENT**

```sql
ELEMENT (SELECT p
  FROM projects p
  WHERE projectID = 4711)
```

- Type casting for type hierarchies

```sql
SELECT (Employee) a
FROM apprentices a;
```
Object creation by means of constructor call

```
SELECT Component(
    productNr: p.productNr,
    title: p.title,
    status: p.status)
FROM products p
WHERE status = "active";
```

**FROM-clause**

- Class extension, collection-valued attribute or reference, result of a method call, or sub-query
- Automatic conversion to bag

```
SELECT e.name
FROM (SELECT p.managed_by
FROM projects p
WHERE p.status = 'finished') e
WHERE e.salary > 9000
```
ODMG OQL: Path Expressions

- Inner path steps can not be multi-valued, e.g. the following is not possible:

```
proj.participants.name;
```

Instead:

```
SELECT e.name
FROM proj.participants e
```

- Multiple multi-valued references

```
SELECT prod.name, prod.cost
FROM employees.participates_in proj,
     proj.develops prod
```

ODMG OQL: Further Concepts

- Arbitrary method calls → problem of side effects
- SQL-like functionality
  - Joins (can often be replaced by following path references)
  - Grouping and aggregate functions
  - Sorting → result is of list type
  - exists and all quantifiers in predicates, e.g.

```
EXISTS p IN projects:
    p.status = 'finished'
```

- Named queries as simple view concept
ODMG Language Bindings

- Exist for C++, Smalltalk, and Java
- Include language-dependent ODL (requires property file for DB specific aspects, such as extents, keys, referential integrity)
- Offer API and type mappings
  - Type mapping for basic data types
  - Type mappings for ODL type constructors (collections, struct, references)
  - Type mapping for pre-defined object types (Date, String)
  - Mapping of exception classes
  - Basic API for accessing database functionality (database connections, queries, transactions)
  - Schema access (catalog, reflection API)

ODMG Java Binding

- Mapping of data types

<table>
<thead>
<tr>
<th>ODL</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>int</td>
</tr>
<tr>
<td>Float</td>
<td>float</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Long</td>
<td>int</td>
</tr>
<tr>
<td>Date</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>Time</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
ODMG Java Binding /2

- Mapping of collection classes in `org.odmg`

<table>
<thead>
<tr>
<th>ODL</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>DSet</td>
</tr>
<tr>
<td>Bag</td>
<td>DBGag</td>
</tr>
<tr>
<td>Array</td>
<td>DArray</td>
</tr>
<tr>
<td>Map</td>
<td>DMap</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- In C++ template classes with prefix `d_`, e.g.

```java
template<class T> class d_Set :
  public d_Collection<T> { ... }
```

ODMG Java Binding /3

- API also in `org.odmg`

- Interfaces
  - Implementation (object factory)
  - Database
  - Transaction
  - OQLQuery

- Offer method declarations for minimum functionality

- Furthermore: exception classes