Object-Oriented Databases

Database Systems
Connolly/Begg/Strachan
Ch. 21/22

Fundamentals of Database Systems
Elmasri/Navathe
Ch. 11/12
Overview

- Why object-oriented databases?
  - Advanced Database Applications
  - Weaknesses of Relational DBMSs

- Core Processes in OO Systems
  - Abstraction
  - Encapsulation

- Elements of an OO System
  - Objects
    - Object Identity
    - Methods
    - Classes

- Relationships in OO Systems
  - Inheritance
    - Polymorphism
  - Complex Objects
    - Relationships between Objects
Advanced Database Applications

- **Examples**
  - Computer-Aided Design
  - Computer-Aided Manufacturing
  - Computer-Aided Software Engineering
  - Office Automation
  - Computer-Aided Publishing
  - Scientific Applications

- **Characteristics**
  - Complex data types
  - Complex relationships
  - Few instances, many data types
  - Many alternative views and versions
  - Dynamic design

- Modern database applications use data in a wide variety of ways.
- Databases are increasingly being used to aid the design and manufacture of products in many industries. For example, the Boeing 777 plane was designed and tested using computer-aided design tools.
- These applications have certain common characteristics that require special data management tools:
  - The applications use *complex data types*. That is, they must store complex structures such as design drawings, video sequences or complex documents.
  - There are *complex relationships* between the data elements. For example, a motor car contains many different parts that are related in many different ways.
  - The applications typically have *many different data types* but very few *data instances*. For example, a plane has many different parts but each part may only occur once.
  - In design applications it is important to be able to work with *many different versions* of the design. A database must be able to monitor and track changes in complex objects.
  - There are frequent *changes* to the data as designs are developed.

*Ref: Connolly, sec 21.1.*
## Weaknesses of Relational DBMSs

<table>
<thead>
<tr>
<th>Weakness</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real World Entities</td>
<td>Relations often do not represent real world entities.</td>
</tr>
<tr>
<td>Semantic Overloading</td>
<td>All relationships are represented by primary/foreign keys.</td>
</tr>
<tr>
<td>Complex Data Types</td>
<td>RDBMSs have provided limited data types.</td>
</tr>
<tr>
<td>Impedance Mismatch</td>
<td>SQL is declarative but programming languages are procedural.</td>
</tr>
<tr>
<td>Static Schema</td>
<td>Relational schemas are normally difficult to change.</td>
</tr>
<tr>
<td>Declarative Queries</td>
<td>Queries state the required result.</td>
</tr>
</tbody>
</table>

- The relational data model was created to provide a model of data. It provides a means of describing and accessing data.
- The relational data model has proved very successful in traditional business situations, for example, record keeping, transaction processing, simple data retrieval.
- However, in more complex situations the relational data model and relational DBMSs have some limitations:
  - Designing relations using normalisation does not always lead to a ‘natural’ entity being represented in each relation. Representing real world entities in the data structure is desirable when the user interacts directly with the data.
  - The relational model represents all relationships using primary and foreign keys. Often it is desirable to distinguish between types of relationship, for example, an ‘is part of’ relationship is very different from an ‘is a’ relationship.
  - RDBMSs have normally provided methods of storing simple data types such as integers and strings, but, until recently, they could not store complex data types such as images or sound.
  - Other weaknesses include limitations in accessing data through SQL, difficulties in changing the structure of the database and limited querying methods.

*Ref: Connolly, sec. 21.2.*
Main Advantage of Object DBMS

Objects as used in program

One-to-One Correspondence

Objects as stored in database
### Overview

<table>
<thead>
<tr>
<th>Overview</th>
<th>Elements of an OO System</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Why object-oriented databases?</td>
<td>• Objects</td>
</tr>
<tr>
<td>– Advanced Database Applications</td>
<td>• Object Identity</td>
</tr>
<tr>
<td>– Weaknesses of Relational DBMSs</td>
<td>• Methods</td>
</tr>
<tr>
<td>• Core Processes in OO Systems</td>
<td>• Classes</td>
</tr>
<tr>
<td>– Abstraction</td>
<td></td>
</tr>
<tr>
<td>– Encapsulation</td>
<td></td>
</tr>
<tr>
<td>• Relationships in OO Systems</td>
<td></td>
</tr>
<tr>
<td>– Inheritance</td>
<td>• Polymorphism</td>
</tr>
<tr>
<td>– Complex Objects</td>
<td>• Relationships between Objects</td>
</tr>
</tbody>
</table>
Abstraction

“The process of identifying the essential aspects of an entity and ignoring the unimportant properties.”

Connolly et al

- Elmasri et al describes abstraction as the process by which we “identify common properties and important aspects of objects in the [world] while suppressing insignificant differences and unimportant details”.

- In database modelling and implementation, abstraction is the process of identifying and describing the structure of data while ignoring how the data will be implemented in a DBMS.

- For example, the entity-relationship model allows the structure of data to be described in terms of entities and relationships between entities without having to be concerned about how the entities or relationships will be constructed using a particular DBMS.

- Abstraction allows the designer to delay making decisions about how the data should be stored until the data’s structure and usage is understood.

Ref: Elmasri, sec 11.1; Connolly, sec 21.3.
Encapsulation

- “The concept of encapsulation, or information hiding, means that we separate the external aspects of an object from its internal details, which are hidden from the outside world.” (Connolly et al)
- Encapsulating an object hides the internal workings of the object so that external observers cannot see how the object carries out its tasks. The object provides a set of operations that it will use to interact with the world.
- Encapsulation is useful because as long as the externally visible operations (the interface) remain the same, the internal workings (the implementation) may be changed. Hence, the implementation of the object is separated from the use of the object.
- In the relational model encapsulation is provided by logical data independence. Data in a relational database is stored in and accessed through relations. The implementation of a particular relation is decided by the database designer.

Ref: Elmasri, sec 11.1; Connolly, sec 21.3.
## Overview

<table>
<thead>
<tr>
<th>Why object-oriented databases?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced Database Applications</td>
</tr>
<tr>
<td>Weaknesses of Relational DBMSs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core Processes in OO Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
</tr>
<tr>
<td>Encapsulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Elements of an OO System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
</tr>
<tr>
<td>Object Identity</td>
</tr>
<tr>
<td>Methods</td>
</tr>
<tr>
<td>Classes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationships in OO Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
</tr>
<tr>
<td>Polymorphism</td>
</tr>
<tr>
<td>Complex Objects</td>
</tr>
<tr>
<td>Relationships between Objects</td>
</tr>
</tbody>
</table>
Objects

- Connolly et al defines an object as “a uniquely identifiable entity that contains both the attributes that describe the state of a real-world object and the actions that are associated with a real-world object”.

- An object has two parts:
  1. **State** The state of an object is given by the values of its attributes.
  2. **Behaviour** The behaviour of an object is determined by its actions.

- An entity has only state information. It does not store information about the actions it can perform or that may be performed upon it.

- Storing data in an object implies that the operations upon the object are also stored with the object. Traditional database theory has separated the use of data from the description and storage of data.

*Ref: Connolly, sec 21.3.2; Elmasri, sec 11.2-11.3.*
The actions that may be performed by an object are called the *methods*.

A method defines a particular behaviour that an object exhibits.

Some methods are hidden (private) and are only visible to other methods in the object.

Other methods are visible (public) and are visible to anyone using the object. Visible methods are described in the interface to the object.

Objects communicate with each other by calling the methods of an object using messages. A message is simply a request from one object to another asking for a method to be executed.

*Ref: Connolly, sec 21.3.4; Elmasri, sec 11.2-11.3.*
Object Identity

- All objects have a unique identifier.
- Called the *Object Identifier* (*OID*).
- Advantages
  - Efficient
  - Fast
  - Independent of Content
  - Invisible

All objects in an object-oriented database have a unique identifier, called the *Object Identifier* (*OID*). The OID of an object never changes and is unique to one object *across the whole object system*.

The OID is not the same as a primary key.
- A primary key changes when the content of the attributes making up the key change. Hence, the primary key is dependent on the current state of the object. A primary key is unique within a single relation.

OIDs are:
- **Efficient** They use very little storage space, unlike primary or foreign keys.
- **Fast** They are fast because they are internal pointers to the objects and not logical pointers that must be translated before they may be used.
- **Independent of Content** An OID is not related to the current state of the object. Two objects may have the same state (the same attribute values) but they will have different OIDs.
- **Invisible** The user does not use the OID.

Note that because the OID is independent of content it is possible for two objects to be indistinguishable. Therefore, a primary key may still be required.

Ref: Connolly, sec 21.3.3; Elmasri, sec 11.2.1.
Classes

- Objects are grouped into classes.
- A class is an abstraction of a group of objects, called *classification*.
- A class has:
  - A name
  - A set of attributes
  - A set of methods
  - A set of constraints

In a database it is necessary to group objects into sets based on their common properties.

*Classification* is the abstraction process which groups objects according to their common properties.

Classification produces a *class*.

A class is a description of an object that describes the common attributes, methods and constraints of objects belonging to the class.

For example, the class *employee* may have the attributes *name* and *address* and the methods *add employee* and *delete employee*. Each object that belongs to the class *employee* will have a name and address and will respond to messages instructing it to add or delete employees.

A class is similar to an *entity type* in the entity-relationship model. An entity type describes the attributes of entities belonging to a particular type.

Ref: Connolly, sec 21.3.5; Elmasri, sec 11.1-11.3.
# Overview

- Why object-oriented databases?
  - Advanced Database Applications
  - Weaknesses of Relational DBMSs
- Core Processes in OO Systems
  - Abstraction
  - Encapsulation

- Elements of an OO System
  - Objects
    - Object Identity
    - Methods
    - Classes

- Relationships in OO Systems
  - Inheritance
    - Polymorphism
  - Complex Objects
    - Relationships between Objects
Inheritance

- Frequently it is desirable to create objects that are similar to other objects but which are not identical.
- For example, a sales staff object is similar to a staff object - they are both staff - but sales staff is a special type of staff. The sales staff object ‘is a’ staff object with additional properties.
- Object-oriented systems use inheritance to implement the ‘is a’ relationship between classes and objects. A class of objects may inherit many of its properties from another class of objects.
- A class of objects that inherits from another class is called a subclass. Subclasses are formed by the abstraction process called specialisation.
- A class of objects that is inherited is called a superclass. Superclasses are formed by the abstraction process called generalisation.
- In the above example, sales staff is the subclass of the superclass staff - sales staff is a staff.
- Sales staff will include all the attributes and methods of staff and may include additional attributes and methods.
- It is possible for a class to inherit from more than one class, called multiple inheritance. In this case it is important to have a mechanism for dealing with conflicts that occur when two superclasses have the same attributes or methods.

Ref: Connolly, sec 21.3.6; Elmasri, sec 11.4.1.
Polymorphism

- Polymorphism means “having many forms”.
- It is the ability to treat objects belonging to different classes as if they belong to the one class.
- For instance, in the example above, all staff objects have an add_bonus() method that adds 10% to their salary. Polymorphism has been used to override the add_bonus() method in staff objects to add 20% to manager bonuses.
- In the following method year_end_updates() a staff object is passed as a parameter and the add_bonus() method is called to update the bonus for the staff memory:

```
method year_end_updates ( var s : staff )
{
    s.add_bonus();
}
```

This method will add the bonus for both sales staff and managers as they are both inherited from staff but it will call different add_bonus() methods depending on the type of staff object provided through s.
- This is possible because the sales staff class is inherited from the staff class and uses the add_bonus() method from the staff class. However, the manager class replaces (overrides) the add_bonus() method with its own add_bonus() method.

Ref: Connolly, sec 21.3.8.
Complex Objects

An object may contain other objects using an ‘is part of’ relationship, e.g. engine is part of car.

- In many of the applications that object-oriented databases are used, it is necessary for one object to contain another.
- It is also possible for one object to contain a set of other objects.
- In object-oriented systems this is achieved using the ‘is part of’ relationship based on the abstraction process called aggregation.
- Aggregation is the process of grouping classes or attributes together to form a new class.
- In the example above, a car object contains an engine object. The engine object is part of the car.
- Structured complex objects contain well-defined components, for example, a car object might consist of engine, body and wheel objects.
- Unstructured complex objects contain large sets of data that are unknown to the DBMS. For example, an image is an unstructured complex object because its components (e.g. elements in the picture) are not represented by objects in the DBMS.

Ref: Connolly, sec 21.3.9; Elmasri, sec 11.5.
In addition to the ‘is part of’ relationship, there are two other methods of representing relationships in an object-oriented system:

1. **Associations** An association is a reference in one object to another object. *Association* is a form of abstraction that relates two or more objects together.

   Associations are formed by storing a *reference attribute* (similar to a foreign key) in one object that references another object. In the example above, the *car* object will contain an attribute *owner* that will point to the object representing the owner of the car.

2. **Messages** The second method of relating two objects is by allowing one object to send a message to the second object. Messages are sent to an object by calling one of its methods.

   - Associations are explicitly represented in the structure of the database and so they are easily identifiable in the database.
   - Message passing is embedded in a method and so it is not easily identifiable in the database.

*Ref: Connolly, sec 22.7.2.*
# Overview

- **Relational Vs Objects**
  - Concepts
  - Programming
  - Database Design
    - Relational
    - Object-oriented
    - Example
      - Entity-relationship
      - Relational
      - Object-oriented

- **Querying an OO Database**
- **Advantages and Disadvantages**
- **Redundancy in OODBs**
  - Normalisation
  - ONF
Relational/Object - Comparison

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Object Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>Class</td>
</tr>
<tr>
<td>Tuple</td>
<td>Instance Object</td>
</tr>
<tr>
<td>Column</td>
<td>Attribute</td>
</tr>
<tr>
<td>StoredProcedure</td>
<td>Method</td>
</tr>
</tbody>
</table>

Accessing Data in an Object DB

- Mixes
  - Declarative (Non-Procedural)
    - Write queries
      - SQL is declarative (-ish)
  - Navigational (Procedural)
    - Move between objects
Querying an OO Database (OQL)

Show the titles of papers by ‘Smith’

```
SELECT p.title
FROM a IN authors, p IN papers
WHERE a.author_name = ‘Smith’
AND p IN a.papers;
```

‘Joining’ using the set attribute.

Ref: Connolly, sec 22.8.5.
Accessing DB Objects from Java

```java
import org.odmg.*;
import java.util.Collection;

Implementation impl = new com.vendor.odmg.Implementation();
Database db = impl.newDatabase();
Transaction txn = impl.newTransaction();

db.open("addressDB", Database.OPEN_READ_WRITE);
txn.begin();

// perform query
OQLQuery query = new OQLQuery("select x from Person x where x.name = \"Doug Barry\"\";\\nCollection result = (Collection) query.execute();
Iterator iter = result.iterator();

while ( iter.hasNext() )
{
    Person person = (Person) iter.next();
    person.address.street = "13504 4th Avenue South";
}

txn.commit();
db.close();
```
Creating an DB Object in Java

```java
person = new Person();
person.ssan = "9999999999";
person.name = "Doug Barry";
person.address.street = "13504 4th Avenue South"
```
Object-Oriented Design

- Relational Design
  1. Identify entities/attributes
  2. Resolve many-to-many relationships
  3. Translate entities into relations
  4. Create primary/foreign key relationships
  5. Implement relations

- Object-Oriented Design
  1. Identify objects/attributes
  2. Identify operations on objects
  3. Establish interface for each object
  4. Implement objects

- The main differences between relational database design and object-oriented database design include:
  - Many-to-many relationships must be removed before entities can be translated into relations. Many-to-many relationships can be implemented directly in an object-oriented database.
  - Operations are not represented in the relational data model. Operations are one of the main components in an object-oriented database.
  - In the relational data model relationships are implemented by primary and foreign keys. In the object model objects communicate through their *interfaces*. The interface describes the data (attributes) and operations (methods) that are visible to other objects.
Example - ER Model

Original ER

Resolved ER

Relations

author(author_name, nationality, date_of_birth)
paper(pno, title, journal, vol, issue, year)
writes(author_name, pno)

- Implementing the original entity-relationship model as a set of relations involves:
  1. Resolving the many-to-many relationships by introducing a new entity. This is necessary because relationships in the relational model are represented by primary keys and foreign keys which are one-to-many relationships.
  2. Adding primary keys and foreign keys.
- The main disadvantage of the relational representation of this model is that the relationship between paper and author is divided between three relations.
  - The contents of the paper relation do not indicate that each paper is written by one or more authors.
- It is important to enforce referential integrity between the relations, for example, all papers must have an author. This is enforced by declaring the primary key and foreign key relationships.
- The operations that may be performed on this data are not represented in the database.
Object-Oriented Implementation

```plaintext
class Author
  properties
    author_name : string;
    nationality : NationalityType;
    date_of_birth : Date;
    papers : Set(Paper)
      inverse is Paper.authors;
  operations
    create(...) delete(...) end Author.

class Paper
  properties
    title : string;
    journal : JournalType;
    vol : number;
    issue : number;
    year : number
    authors : Set(Author)
      inverse is Author.papers;
  operations
    create(...) submit(...) end Paper.
```

- In the object-oriented implementation, the relationship between *author* and *paper* is represented using the `Set` type.
  - The definition:
    ```plaintext
papers : Set(Paper)
    inverse is Paper.authors;
    ```
    declares an attribute *papers* that contains a set of *paper* objects. The `inverse` command states that objects of type *paper* will contain a set of *author* objects.
  - Referential integrity is enforced by the `inverse` statement.
- The object model more closely resembles the entity-relationship model.
- The relationship between *author* and *paper* is represented by the `set` attributes in each class.

*Ref: Connolly, sec 22.8.3.*
## Overview

- **Relational Vs Objects**
  - Concepts
  - Programming
  - Database Design
    - Relational
    - Object-oriented
    - Example
      - Entity-relationship
      - Relational
      - Object-oriented

- **Querying an OO Database**

- **Advantages and Disadvantages**

- **Redundancy in OODBs**
  - Normalisation
  - ONF
Advantages and Disadvantages

- **Advantages**
  - Enriched modelling
  - Extensibility
  - No impedance mismatch
  - Expressive query language
  - Advanced applications

- **Disadvantages**
  - Lack of single data model
  - Little experience
  - Few standards
  - Query optimisation is difficult
  - Complex
## Overview

- **Relational Vs Objects**
  - Concepts
  - Programming
  - Database Design
    - Relational
    - Object-oriented
    - Example
      - Entity-relationship
      - Relational
      - Object-oriented

- **Querying an OO Database**

- **Advantages and Disadvantages**

- **Redundancy in OODBs**
  - Normalisation
  - ONF
Object Class vs Relation

- Non-1NF
  - Object attributes can be complex
  - Object attributes can be multi-valued

- Non-2NF/3NF
  - Objects are identified by the OID not by a key
Object Class

Class Employee {
    string ssn
    string name
    integer salary
    string dept_name
    integer dept_budget
    set <string> dept_locations
}

Update anomalies on dept_name, dept_budget, dept_locations
Update Anomalies

Class Employee {
    string ssn
    string name
    integer salary
    string dept_name
    integer dept_budget
    set <string> dept_locations
}

- Deletion anomaly
  - Cannot delete a department without deleting all employees in the department.

- Modification anomaly
  - Changing a department's details requires all the objects in employee to be changed.
Object Class Dependencies

Class Employee {
    string ssn
    string name
    integer salary
    string dept_name
    integer dept_budget
    set <string> dept_locations
}

"Give two attributes X and Y that belong to class C, Y is object functionally dependent on X if and only if the value of Y is determined uniquely for each value of X"
Normalising a Class

Class Department {
    string dept_name
    integer dept_budget
    set <string> dept_locations
}

Class Employee {
    string ssn
    string name
    integer salary
    Department dept
}

“An object class C is said to be in the object normal form if and only if all determinants of object functional dependencies are the object identifier (oid) of the class C.”