2 Object-Oriented Databases

2.1 Motivation

The relational model is the basis of many commercial relational DBMS products (e.g., DB2, Informix, Oracle, Sybase) and the structured query language (SQL) is a widely accepted standard for both retrieving and updating data.

The basic relational model is simple and mainly views data as tables of rows and columns. The types of data that can be stored in a table are basic types such as integer, string, and decimal.

Relational DBMSs have been extremely successful in the market. However, the traditional RDBMSs are not suitable for applications with complex data structures or new data types for large, unstructured objects, such as CAD/CAM, Geographic information systems, multimedia databases, imaging and graphics. The RDBMSs typically do not allow users to extend the type system by adding new data types. They also only support first-normal-form relations in which the type of every column must be atomic, i.e., no sets, lists, or tables are allowed inside a column.

Due to the new needs in database systems, a number of researches for OODBMS have begun in the early 80's.

2.2 Concept & Features

While a relational database system has a clear specification given by Codd, no such specification existed for object-oriented database systems even when there were already products in the market. A consideration of the features of both object-oriented systems and database management systems has lead to a definition of an object-oriented database, which was presented at the First International Conference on Deductive, and Object-oriented Databases in the form of a manifesto in 1989. This 'manifesto' distinguishes between the mandatory, optional and open features of an object-oriented database.

The mandatory features, which must be present if the system is to be considered (in the

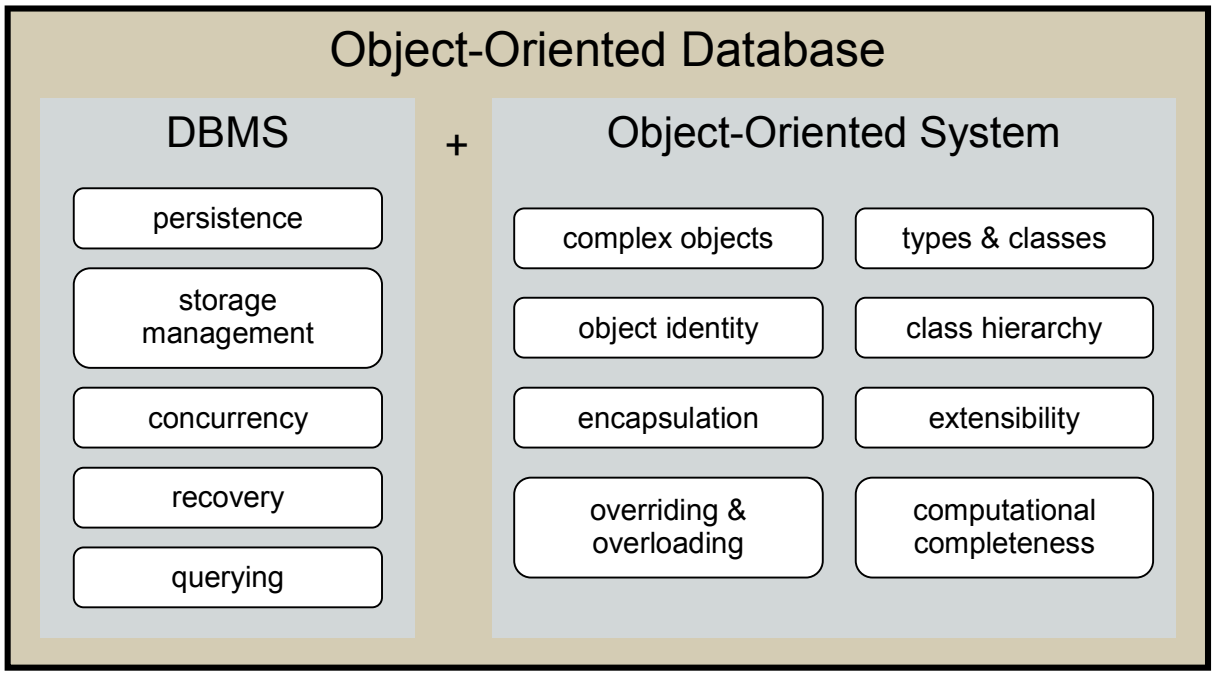


 Figure 1 OODB feathues

opinion of the manifesto authors) to be an object-oriented database, are defined in the following two paragraphs. The first part describes features of object-oriented system, as the second part features of database system.

2.2.1 Mandatory features of object-oriented systems

Support for complex objects
A *complex object* mechanism allows an object to contain attributes that can themselves be objects. In other words, the schema of an object is not in first-normal-form. Examples of attributes that can comprise a *complex object* include lists, bags, and embedded objects.
Object identity
Every instance in the database has a unique identifier (OID), which is a property of an object that distinguishes it from all other objects and remains for the lifetime of the object. In object-oriented systems, an object has an existence (identity) independent of its value.
Encapsulation
Object-oriented models enforce *encapsulation* and *information hiding*. This means, the state of objects can be manipulated and read only by invoking operations that are specified within the type definition and made visible through the **public** clause.
In an object-oriented database system encapsulation is achieved if only the operations are visible to the programmer and both the data and the implementation are hidden.
Support for types or classes
- *Type*: in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. In programming languages *types* can be used at compilation time to check the correctness of programs.
- *Class*: The concept is similar to type but associated with run-time execution. The term *class* refers to a collection of all objects with the same internal structure (attributes) and methods. These objects are called instances of the *class*.
- Both of these two features can be used to group similar objects together, but it is normal for a system to support either *classes* or *types* and not both.

Class or type hierarchies
Any *subclass* or *subtype* will inherit attributes and methods from its superclass or supertype.
Overriding, Overloading and Late Binding
- *Overloading:* A class modifies an existing method, by using the same name, but with a different list, or type, of parameters.
- *Overriding*: The implementation of the operation will depend on the type of the object it is applied to.
- *Late binding*: The implementation code cannot be referenced until run-time.

Computational Completeness
SQL does not have the full power of a conventional programming language. Languages such as Pascal or C are said to be computationally complete because they can exploit the full capabilities of a computer. SQL is only relationally complete, that is, it has the full power of relational algebra. Whilst any SQL code could be rewritten as a C++ program, not all C++ programs could be rewritten in SQL.

For this reason most relational database applications involve the use of SQL embedded within a conventional programming language. The problem with this approach is that whilst SQL deals with sets of records, programming languages tend to work on a record at a time basis. This difficulty is known as the impedance mismatch. Object-oriented databases attempt to provide a seamless join between program and database and hence overcome the impedance mismatch. To make this possible the data manipulation language of an object-oriented database should be computationally complete.

2.2.2 Mandatory features of database systems

A **database** is a collection of data that is organized so that its contents can easily be accessed, managed, and updated. Thus, a database system contains the five following features:

Persistence
As in a conventional database, data must remain after the process that created it has terminated. For this purpose data has to be stored permanently on secondary storage.
Secondary Storage Management
Traditional databases employ techniques, which manage secondary storage in order to improve the performance of the system. These are usually invisible to the user of the system.
Concurrency
The system should provide a concurrency mechanism, which is similar to the concurrency mechanisms in conventional databases.
Recovery
The system should provide a recovery mechanism similar to recovery mechanisms in conventional databases.
Ad hoc query facility
The database should provide a high-level, efficient, application independent query facility. This needs not necessarily be a query language but could instead, be some type of graphical interface.

The above criteria are perhaps the most complete attempt so far to define the features of an object-oriented database in 1989. Further attempts to define an OODBS standard were made variables of researchers. One of them is a group called Object Data Management Group (ODMG). They have worked on an OODBS standard for the industry. The recent release is ODMG-2 in1997.

2.3 Making OOPL a Database

Basically, an OODBMS is an object database that provides DBMS capabilities to objects that have been created using an object-oriented programming language (OOPL). The basic principle is to add persistence to objects and to make objects persistent. Consequently application programmers who use OODBMSs typically write programs in a native OOPL such as Java, C++ or Smalltalk, and the language has some kind of Persistent class, Database class, Database Interface, or Database API that provides DBMS functionality as, effectively, an extension of the OOPL.

Object-oriented DBMSs, however, go much beyond simply adding persistence to any one object-oriented programming language. This is because, historically, many object-oriented DBMSs were built to serve the market for computer-aided design/computer-aided manufacturing (CAD/CAM) applications in which features like fast navigational access, versions, and long transactions are extremely important. Object-oriented DBMSs, therefore, support advanced object-oriented database applications with features like support for persistent objects from more than one programming language, distribution of data, advanced transaction models, versions, schema evolution, and dynamic generation of new types.

The following subsection describes object data modeling and the persistency concept in OODB.

2.3.1 Object data modeling

An object consists of three parts: structure (attribute, and relationship to other objects like aggregation, and association), behavior (a set of operations) and characteristic of types (generalization/serialization). An object is similar to an entity in ER model; therefore we begin with an example to demonstrate the structure and relationship.

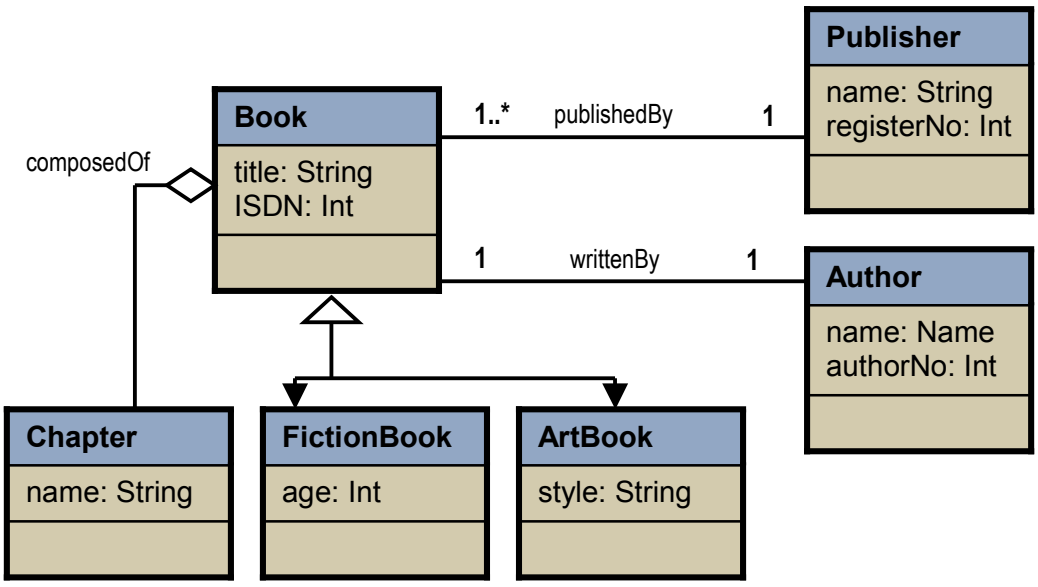


Figure 2 Book example

The structure of an object `Book` is defined as following:

```
class Book {
      title: String;
      ISDN: Int;
      publishedBy: Publisher inverse publish;
      writtenBy: Author inverse write;
      chapterSet: Set<Chapter>;
}

class Author {
      name: String;
      authorNo: Int;
      write: Book inverse writtenBy;
}
```

Attributes are like the fields in a relational model. However in the `Book` example we have, for attributes `publishedBy` and `writtenBy`, complex types `Publisher` and `Author`, which are also objects. Attributes with complex objects, in RDNS, are usually other tables linked by keys to the employee table.

Relationships: `publish` and `writtenBy` are associations with I:N and 1:1 relationship; composed_of is an aggregation (a Book is composed of chapters). The 1:N relationship is usually realized as attributes through complex types and at the behavioral level. For example,

```
class Publisher {
      …
      publish: Set<Book> inverse publishedBy;
      …
Method insert(Book book){
      publish.add(book);
}
```

Generalization/Serialization is the `is_a` relationship, which is supported in OODB through class hierarchy. An `ArtBook` is a `Book`, therefore the `ArtBook` class is a subclass of `Book` class. A subclass inherits all the attribute and method of its superclass.

```
class ArtBook extends Book {
        style: String;
}
```

Message: means by which objects communicate, and it is a request from one object to another to execute one of its methods. For example:

```
Publisher_object.insert ("Rose", 123,…)
```

i.e. request to execute the `insert` method on a `Publisher` object **)**

Method: defines the behavior of an object. Methods can be used

– to change state by modifying its attribute values

– to query the value of selected attributes

The method that responds to the message example is the method `insert` defined in the `Publisher` class.

2.3.2 Persistence of objects

Persistence, as mentioned before, means that certain program components "survive" the termination of the program. Thus these components have to be stored permanently on secondary storage.

Typically, persistence or non-persistence is specified at object creation time. There are two possible ways to make an object persistent:

(1) explicitly call built-in function ***persistence*** – certain objects are persistent

(2) automatically make object of persistent types persistent – all objects are persistent

There are several object-oriented DBMSs in the market (e.g., Gemstone, Objectivity/DB, ObjectStore, Ontos, O2, Itasca, Matisse). These products all support an object-oriented data model. Specifically, they allow the user to create a new class with attributes and methods, have the class inherit attributes and methods from superclasses, create instances of the class each with a unique object identifier, retrieve the instances either individually or collectively and load and run methods.

Most of these OODBs support a unified programming language and database language. That is, one language (e.g., C++ or Smalltalk) in which to do both general-purpose programming and database management.

2.4 GemStone

The GemStone data management system, developed at Servio Logic, was one of the first and simplest commercial OODBMS products. It is based on Smalltalk, with very few extensions. GemStone merges object-oriented language concepts with those of database systems. And provides an object-oriented database language called OPAL which is used for data definition, data manipulation and general computation.

2.4.1 Architecture

The GemStone system exhibits *client/server* architecture, and is distributed over two processes: the *Gem* and the *Stone* processes. The Stone process, running on the server,

delivers the data management capabilities performing disk I/O, concurrency control, recovery and authorization.

Stone uses unique object id called object-oriented pointers (OOPs) to refer to objects, and an object table to map an OOP to a physical location. The Gem process runs either on the server or on a client. It comprises compilation facilities, browsing capabilities, and user authentication.

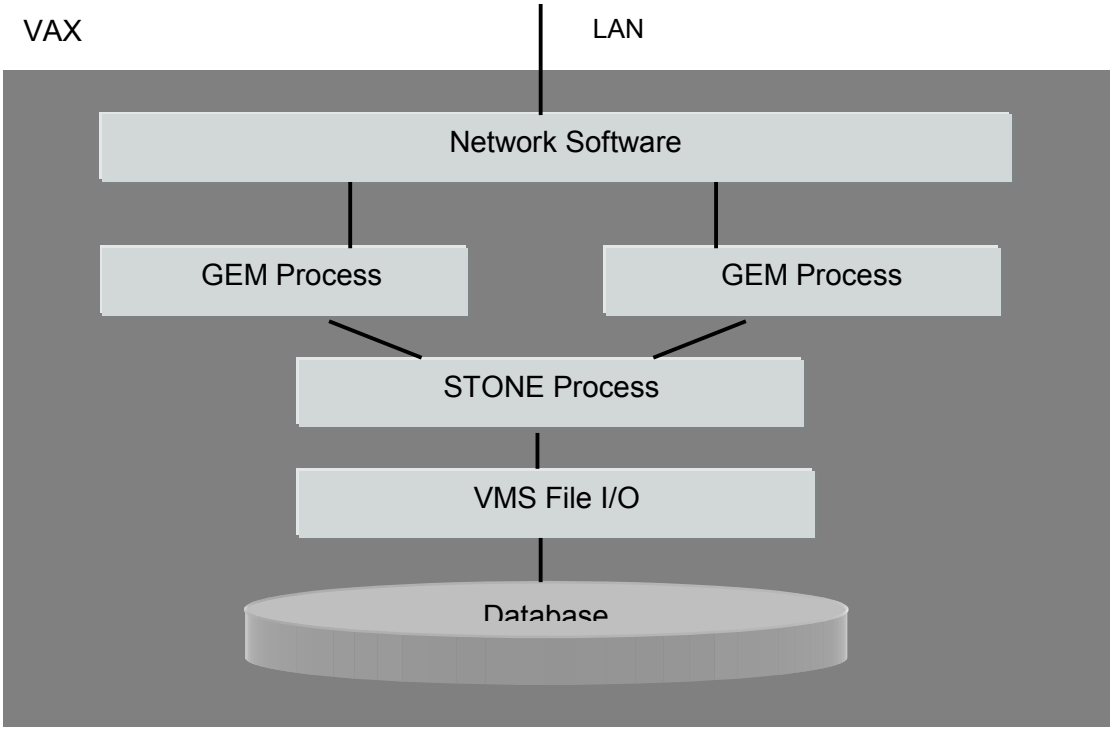


Figure 3 GemStone Architecture

2.4.2 Object model

The GemStone object model is very closely related to the Smalltalk-80 model.
The three principal concepts of the GemStone model and language are *object*, *message*, and *class*. All the objects in GemStone are made persistent.

2.4.2.1 Classes

Every GemStone object is an instance of exactly one class. Objects with the same internal structure and methods are grouped together into a class and are called instances if the class.

2.4.2.2 Objects

An object is a chunk of private memory with a public interface. Internally, most objects are divided into fields called *instance variables*. Each instance variable can hold a value, which is another object. Objects communicate with other objects by passing messages. Object is the root of all super/subclass hierarchy.

2.4.2.3 Messages

In GemStone, all actions are invoked by message passing. Messages are requests for the receiving object to change its state or return a result. The set of messages an object responds to is called *protocol* (its “public interface”). An object may be inspected or changed only through its protocol. The basic form of all message expressions is <receiver> <message>. The <receiver> part is an identifier or expression denoting an object that receives and interprets the message. The <message> part gives the selector of the message and possible arguments to the message.

2.4.2.4 Methods

The methods are the concrete implementations that can be invoked by a message sent to an instance. An object can only responds to a message if it contains a method with a selector that matches the message format. Methods are provided to query and manipulate the internal structure.

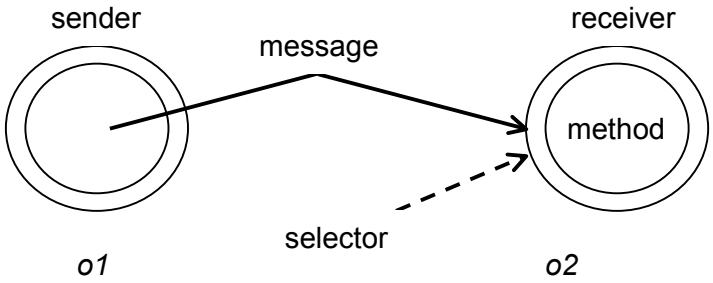


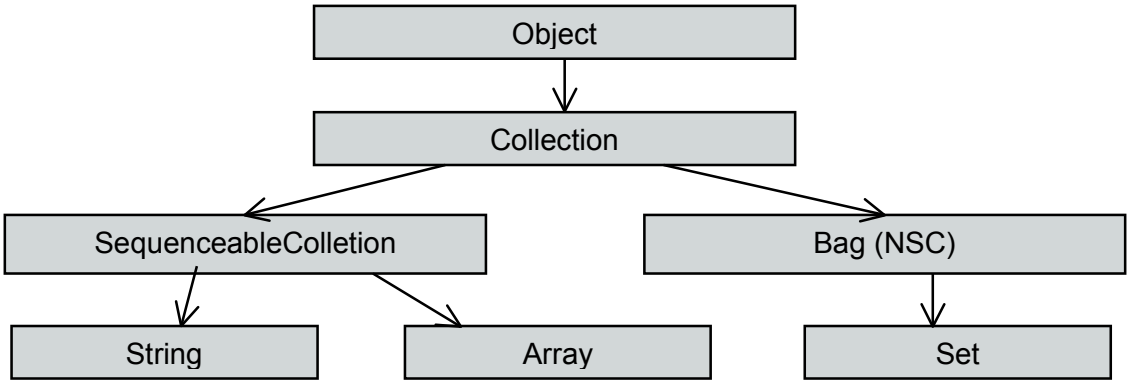
Figure 4 Message passing in GemStone

2.4.3 Collection classes

In GemStone, a class defines the structure of its instances, but rarely keeps track of all the instances. Instead, collection objects – Arrays, Bags, sets – can store groups of instances – not necessarily of the same type – in indexable or anonymous storage slots. GemStone provides built-in support for managing collections of objects by the pre-defined *Collection* class and its subclasses.

- Array: like String, is a subclass of Collection's subclass SequenceableColletion.
- Bags and Sets: are *non-sequenceable Collectio*ns, in which instance variables are *anonymous.* They do not maintain any order on their elements. The difference between a Bag and a Set is that the instances of Bag may contain the same object several times, whereas a Set contains an element just once – even though it might have been inserted several times

Figure 5 Class hierarchy of Collection classes



2.5 Comparisons of OODBS & RDBS

2.5.1 Correspondence between OODBS and RDBS

To have an idea about OODBS, the table shows the correspondence between object-oriented and relational database systems:

OODBS	RDBS
object	tuple
instance variable	column, attribute
class hierarchy	database scheme (is-a relation)
collection class	relation
OID	key
message	procedure call
method	procedure body

The correspondence between object-oriented and relational database systems

Note that this correspondence table is only an approximate equivalence. The properties in OODBS are actually not applicable in RDBS and vice versa.

2.5.2 Comparison

Although there are great advantages of using an OODBMS over an RDBMS, some disadvantages do exist. The following table shows the advantages and disadvantages using OODBS over RDBS.

Advantage	Disadvantage
▪ Complex objects & relations ▪ Class hierarchy ▪ No impedance mismatch ▪ No primary keys ▪ One data model ▪ High performance on certain tasks ▪ Less programming effort because of inheritance, re-use and extensibility of code	▪ Schema change (creating, updating…) is non trivial, it involves a system wide recompile. ▪ Lack of agreed upon standard ▪ Lack of universal query language ▪ Lack of Ad-Hoc query ▪ Language dependence: tied to a specific language ▪ Don't support a lot of concurrent users

Advantages and disadvantages using OODBS over RDBS

Because of the existing disadvantages of using OODBS, the approach of ORDBMS has become popular. In the future, it is likely that we will see the continued presence of OODBMS that address the needs of specialized market and the continued prominence of ORDBMSs that address the needs of traditional commercial markets.

The following chapter specifies the indexing design issues and its implementation in GemStone.