A Data Structure Approach to Implement the Characteristic of Inheritance in An Object-Oriented Database Management System

Suh-Ying Wur  
Soochow University

Tse-Wing Lo  
Soochow University

Follow this and additional works at: http://aisel.aisnet.org/pacis1995

Recommended Citation
http://aisel.aisnet.org/pacis1995/100
A Data Structure Approach to Implement the Characteristic of Inheritance in An Object-Oriented Database Management System

Suh-Ying Wur
Soochow University
Taipei, Taiwan, ROC

Tse-Win Lo
Soochow University
Taipei, Taiwan, ROC

Object-Oriented Database Management Systems (OODBMSs) have been discussed and developed greatly during the past several years. Many systems implement the characteristic of inheritance through using an object-oriented programming language, such as Smalltalk or C++, which serves as an external programming interface to the OODBMS. In this paper, we propose a data structure approach to implement the characteristic of inheritance. Based on the proposed data structure approach, we design the algorithms for implementing the schema evolution. It is shown that, based on our approach, the characteristic of inheritance of a class hierarchy remains intact after performing the schema evolution. With our approach, it becomes unnecessary for users to learn or to understand any object-oriented programming language in order to communicate with an OODBMS.

1 Introduction

Recently, some of the OODBMSs, which are either commercial products or laboratory prototypes, have been discussed and compared in several papers like [Hurson et al., 1993] and [Ahmed et al., 1992]. According to these papers, we learn that most of the systems, including GemStone, Ontos/Vbase, ObjectStore [Object Design Inc., 1992], and Versant, use object-oriented programming languages such as Smalltalk or C++ to be the system external interface. This means that a database user has to learn an object-oriented programming language first before he can design or implement a database application. Meanwhile, it is also shown that the characteristic of inheritance, which is one of the most important characteristics of OODBMS, is provided by the object-oriented programming language rather than by the OODBMS. On the other hand, some systems developed their own object-oriented database languages as their external interfaces, for instance, ORION [Kim et al., 1990] [Kim et al., 1989] and Iris.

In this article, we propose a data structure and a file organization to implement the characteristic of inheritance. Based on the proposed approach, database applications could be implemented in an SQL-like language rather than in an object-oriented programming language.

2 Data Structure Design

In this section we propose data structures designed for implementing the characteristic of inheritance.

2.1 Attribute Information Table

The Attribute Information Table is used to store the information for each attribute defined in a class. It consists of the following fields: AttributeName, AttributeType, ValueSize, DefaultValue, and AttributeFileName. The DefaultValue defines the value given for the attribute in the class. This value, if defined, will be inherited to the children under the class. That is to say, all of the instances under this class and its subclasses will have the value defined in the DefaultValue if the instance does not specifically define its own value for the attribute. Meanwhile, there is an AttributeFileName field for each attribute. The AttributeFileName indicates a data file which stores the values specified by instances on that attribute. It means that the values stored in a data file are on an attribute base rather than on a relation base in the relational database model. If all of the instances of the class and its subclasses do not specify any values for that attribute at all, the data file will be empty and the value defined in the DefaultValue will be applied. We suppose that the data file designed on the attribute base is one of the key design concept in our research.

2.2 Index Matrix

An Index Matrix is associated with each class. Each row of the Index Matrix represents an instance created under the class. Each column represents an attribute appeared in the class either defined by the class itself or inherited from the superclasses. The element of the Index Matrix indicates the location of a data file, where the attribute value is stored. If the element is zero, it means that the instance does not specify its own value for the corresponding attribute. Under this situation, the value defined in the DefaultValue in the class or even its superclass will be used as the attribute value for the instance. Hence, the characteristic of inheritance is implemented.

2.3 Collector

Besides data structures described in the previous subsections to represent the attribute information and instance information, we also propose another data structure to implement the inheritance function. The data structure is
a dynamic ordered list which stores pointers pointing to the locations of attributes in the Attribute Information Tables. This dynamic ordered list is called the "collector".

There is a function to create and maintain the collector. When the function is called, it collects every attribute defined in each class from the current class back to the root class according to the class hierarchy. While collecting, if two attributes are defined with the same name, the collector will take the one near to the current class. After being collected, the list of collector points to all of the attributes, including those attributes defined in this class and those attributes inherited from its superclass chain, that the current class has.

3 File Organizations

We design a different kind of file organization, named the StackFile, for the implementation to improve the insertion and deletion operation.

The StackFile is a special file organization. It does not only consist of the regular data storage but also the other file management information such as EOR, TOP, and STACK. The EOR stands for End of Record indicating the location of last datum in the file. The TOP indicates the location of the top of the STACK. The STACK contains those unused locations in the file. The STACK is on the position right after the location of the last datum.

A StackFile is associated to an attribute. The StackFile stores the instance-defined values for the attribute. Therefore, the values in a StackFile could belong to instances under different classes.

The reason why we design the StackFile at this kind of organization is under the concern of frequently manipulating in insertion and deletion. When a datum is going to be inserted, based on our file organization, it will check whether the STACK holds some free locations first. If yes, a free location will be popped from the STACK and then the datum will be stored in that location. While the STACK is empty, the free location will be obtained from the EOR and then the datum is stored in the location. When we delete a datum from the StackFile, the location occupied by the deleted datum will be pushed into the STACK. Therefore, the unused locations can be reused at the next insertions.

4 Creation, deletion, update, and inheritance

In this section, we selectively present the process about manipulating the schema evolution of an object-oriented database for creation, deletion, and update according to our approach. Through the process, we can see that the data structures will keep the characteristic of inheritance intact.

4.1 Create an attribute with value

The process of creating an attribute with value in a specified class is as follows:

Append a new row to the Attribute Information Table of the specified class
Set the AttributeName, AttributeType, ValueSize, and DefaultValue as specified
Create an AttributeFile and store the file name in the field of AttributeFileName
Append a new column in the Index Matrix of the specified class to represent the new attribute
Initialize the value in each element of the new column to be zero
Maintain all of the Index Matrices of the subclasses under the specified class by appending a new column and initializing the value in each element of the new column to be zero

After creation, the subclasses under the specified class should be aware of the newly created attribute eventually due to the characteristic of inheritance. The above process implements this characteristic at the attribute creation time.

4.2 Delete a class

To delete a class is a very serious action, it will reorganize the class hierarchy. Since the specified class is deleted, those attributes and instances specified under this class should be all deleted as well. Besides, its subclasses should not consist of the attributes specified by the deleted class at all. After the specified class is deleted, the position of that class in the class hierarchy will be replaced by its direct subclasses.

The process of deleting a class is as follows:

Delete the AttributeFile associated with each attribute indicated by the AttributeFileName field in the Attribute Information Table
Delete all instances under the specified class
Delete the Index Matrix of the specified class
Maintain all of the Index Matrices of the subclasses under the specified class to delete the columns corresponding to the attributes defined in the specified class but not redefined in the subclasses of the specified class through the inheritance information obtained from the collector operation
Maintain the class hierarchy to link its direct subclasses as the subclasses of the superclass of the specified class
Delete the Attribute Information Table

4.3 Update the attribute's data type in the original class

To update the attribute's data type in the original class means to redefine the attribute. The result will affect all of the instances of the specified class and its subclasses. Since
the attribute's data type is updated, the original data may not represent the attribute anymore and should be deleted as well. However, in some cases such as updating data type from integer to real, the original values are still useful and needed. The situation is happened when the old data type and the new data type are assignment equivalent. No matter what, our approach can handle the above two situations without any problems. We will state the processes for both of them below.

The process of updating the attribute's data type in the original class without assignment equivalent condition is as follows:

- Get the Index Matrix of the specified class
- Reset the values in the column corresponding to the specified attribute
- Perform the following operations for each subclass under the specified class:
  - Call collector function to get the inheritance information about the subclass
  - If the specified attribute in the subclass is not inherited from the specified class, nothing is to be done. Otherwise, maintain the Index Matrix by resetting the values in the column corresponding to the specified attribute
- Update the AttributeType/AttributeSize and give a new AttributeFileName in the Attribute Information Table of the specified class

The process of updating the attribute's data type in the original class with assignment equivalent condition is as follows:

- Get the AttributeFileName from the Attribute Information Table as the old AttributeFileName
- Update the AttributeType/AttributeSize and give a new AttributeFileName in the Attribute Information Table of the specified class
- Perform the following operations for each element in the column corresponding to the specified attribute:
  - Get the element as an location index to the old AttributeFileName
  - Get the attribute value from the old AttributeFile by the location index
  - Assign the old attribute value to be the new data
  - Write the new attribute value to the new given AttributeFileName and return the location as new index
  - Store the new index into the Index Matrix
- Perform the following operations for each subclass under the specified class:

Call collector function to get the inheritance information about the subclass
If the specified attribute in the subclass is not inherited from the specified class, then nothing is to be done. Otherwise, maintain the Index Matrix for the subclass as the same as operations performed for the specified class just described

5 Conclusion
We have implemented an experimental system to show the feasibility of the proposed data structures. The final goal for this system is to be implemented as an OODBMS.

The system is implemented using Borland C++. It is noted that although the system is implemented using C++, it is unnecessary for users to understand C++ in order to communicate with the system. Instead, we provide a Graphic User Interface (GUI) as the system external interface.

Of course, it is still a challenge for us to achieve our final goal. There are several further research issues to be overcome to implement an OODBMS based on our approach. They are (1) transaction management, (2) security control, (3) object clustering, (4) object storage management, (5) SQL interface, and (6) performance measurement.

However, we believe that this data structure and file organization approach will provide users a user-friendly interface to an OODBMS.

References


