Schema Evolution in OODBs Using Class Versioning

Simon Monk and Ian Sommerville
Computing Dept, Lancaster University,
Lancaster, LA1 4YR, UK. {srm | is} @uk.ac.lancs.comp.

Abstract

This paper describes work carried out on a model for the versioning of class definitions in an object-oriented database. By defining update and backdate functions on attributes of the previous and current version of a class definition, instances of any version of the class can be converted to instances of any other version. This allows programs written to access an old version of the schema to still use data created in the format of the changed schema.

1. Introduction

Where a database consists of data and a schema, it is the schema that is usually considered to be stable and the data that changes. However, schemas are not as constant as one would expect. Since the role of a database is to model a part of the real world, if the part that is to be modelled changes in structure, then the database structure (the schema) must change with it. Furthermore, database designers, like all designers, are prone to making mistakes and failing to identify all the data modelling requirements of a database. To correct such mistakes, the schema must be modifiable.

The variability of database systems is illustrated by Sjøberg [13]. Sjøberg documents work carried out to measure the extent of schema evolution in developing and in-use systems. His study is based on a health management system now in use in a number of hospitals. The system automatically recorded changes made to the schema over a period covering its development and initial use. During this time, there was an increase in the number of relations in the system from 23 to 55 and an increase in the number of attributes from 178 to 666. These results illustrate the extent to which schema evolution occurs in practical database applications.

Class modification, class versioning and schema versioning have all been used to support schema evolution. All these methods have disadvantages which are explained below.

Class modification [1, 5, 9, 10] is simply the modification of existing class definitions. Instances of the class that is modified are converted to the format of the modified class definition. Class modification does not support forward compatibility. That is, 'old' programs written to access one version of the schema cannot be guaranteed to continue to work when accessing data created after the schema has been changed. In contrast, backward compatibility is when programs written assuming the latest version of the schema, can still access data that was created under a previous version of the schema. Class modification can achieve a degree of backward compatibility by converting instances created under the old schema version to the new schema, but it cannot address the problem of forward compatibility.

Class versioning [2, 3, 4, 8, 10, 14] is the creation of a new version of the class definition. That is, the class definition before the change, is retained, allowing multiple versions of a class definition to co-exist. Unlike class modification, class versioning supports both forward and backward compatibility.

In this paper, a novel system of class versioning (CLOSQL) using *dynamic instance conversion* is proposed that overcomes some of the limitations of existing class versioning systems.

The remainder of this paper is organised in four sections. Section 2 describes existing systems that support schema evolution. Section 3 introduces the authors' approach to class versioning. Section 4 illustrates the authors' system with an example and finally section 5 concludes the paper.

2. Existing Systems

In this section, existing systems for supporting schema evolution using class versioning and *schema versioning* are surveyed. Systems that use class modification to accomplish schema evolution, such as ORION [5, 6] and GemStone [5, 6], are not described here as it is class versioning that is explored in this paper.

Another approach to schema evolution involves the use of temporal databases [11, 12] that represent temporal information about the data and meta-data.

2.1. Class Versioning

Class versioning is distinguished from class modification in that, in carrying out a change to a class definition, the existing class definition is not changed, but rather a new version of the class definition is created with the required change incorporated. Thus old and new versions of the class definition co-exist. Instances of an older version may be converted into instances of the new class version or they may remain in their current state and *emulate* the latest class definition during queries [9].

Encore

The ENCORE system is a prototype object-oriented database (OODB) that addresses some of the problems of schema evolution [14]. Classes can be versioned, and the set of versions of one class is called the *version set* of the class. In every version set there is one version that is termed the current version and is always the version most recently created.

In addition to a version set for each class, there is also a *version set interface* for the class. This is a virtual class definition, that contains the union of all attributes of the versions of that class. When the class has only one version, the version and the version set interface are identical. As new versions of the class are created, the version set interface is extended to add extra attributes.

Figure 1 shows how ENCORE would represent three versions of the class *Person*.

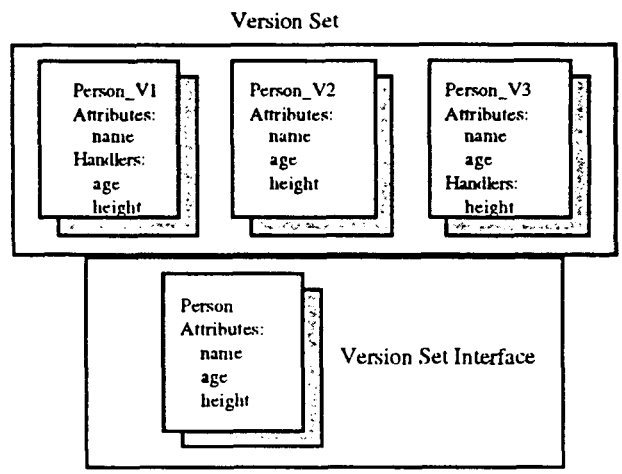

Figure 1. Version Set Interface in ENCORE.

It is the version set interface that is visible to the user, presenting a single view of the version set of the class. Since attributes are never removed from the version set interface, any program written to communicate with the version set interface, will always find a reference to the attribute it requires, even if that attribute does not exist for instances of some versions of the class. For this reason a handler is defined for every attribute of a class version that appears in the version set interface, but not in that version of the class definition. These handlers can return values. When a query is made that refers to an attribute that is not present in some instance, then a default value for the attribute is returned by the handler.

In Figure 1, handlers are defined for *age* and *height* that could for example return 0 as a default value when an attempt is made to access either of these attributes for instances of the class *Person_V1*.

ENCORE has two types of handler, *read handlers* and *write handlers*. The handlers we have considered so far have been *read handlers*. That is they have been invoked as the result of an exception arising during the attempted reading of an attribute. Write handlers are invoked in ENCORE when an attempt is made to write a value to an attribute that is not present in an instance of some version of a class that does not have the attribute required. Write handlers return *True* or *False* to indicate whether the attempt to write was successful or not. The write can only be considered successful (true returned) if the value of the attribute on which the write was attempted happened to be the same value as the default value specified in the read handler for that attribute. In figure 1, if the action of the read handler for the attribute *age* in *Person_V1* was to return the value 0 then the only legal writes to that attribute would be the writing of a value 0.

This highlights a serious limitation of ENCORE. The inability to associate additional storage with existing attributes [4]. This means that although an additional attribute can be defined for a class, only a fixed, read only, default value can be provided for it in the pre-versioned class.

In ENCORE, attributes of the same name but referenced in different versions of a class are assumed to represent the same information in both versions. This means that it is not possible to represent a change in the semantics of the attribute between versions, such as the changing of units of height from cm to inches. Such a change requires the scaling of numeric values during conversion from one version of the class to the other.

AVANCE

The AVANCE project [2, 3], develops general object versioning as part of their prototype OODB. This object versioning is extended to the versioning of class definitions.

The system adopts a similar approach to ENCORE, using exception handling to cope with mismatches between the version of the object expected by the query and the actual version of a instance. The exception handlers service the query with values appropriate to the version of the class demanded by the query.

Clamen

Clamen [4] proposes a scheme for class versioning in OODBs. His approach is based on effectively creating a new version of each instance of a class that has been versioned. So, the creation of a new version of a class results in a new version of every instance of the class. He calls these instance versions *facets*. This is shown in figure 2.

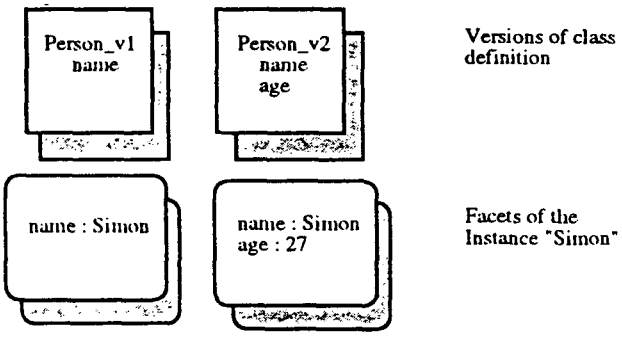

Figure 2. Facets.

The storage requirement for this can be reduced by allowing attributes present in more than one facet to be shared. A user can fill in any of the facets (usually the latest facet) and the other facets should be completed automatically. This automatic completion is deferred, until required. The system is not implemented as yet and the actual mechanism for the automatic completion of values in facets has been left undefined.

2.2. Schema Versioning

OTGen [7] addresses the problem of changing the schema as a whole rather than the piecemeal changing of individual classes. Lerner and Habermann provide this facility in the form of an interactive design tool for use by the database administrator. The output of which is a copy of the whole database, but to a new schema.

This system whilst providing a very powerful and practical tool for allowing databases to evolve forward, does not attempt to address the issue of allowing backwards compatibility, except by the rather extreme step

of converting the entire database back to an earlier version. This system requires the conversion of the entire database, something that is both expensive for large databases, and unnecessary if the schema change is localised.

3. CLOSQL

The approach to schema evolution taken by the author is based on the principle that it is not sufficient to change the structure of the database without any consideration for the underlying semantics of the data. So a simple schema change such as -

change the name of an attribute

- could in reality mean -

replace an attribute with a new attribute with a new name, which means something different to the old attribute, but is loosely related to it in some way.

This can be illustrated with an example. Consider the situation below where the name of the attribute *name* is changed to *surname*.

Before	*After*
Person	*Person*
name	*surname*

At first sight, this may seem to be a relatively straight forward change. Indeed, in most databases, this change would be made at face value. Any instances with a value for the attribute *name*, would have that value associated with *surname* on conversion to or emulation of the new class definition. However *name* and *surname* may not be exact equivalents. Indeed it is quite likely that the reason the attribute's name was changed is that its meaning had shifted. This becomes apparent when the instances of the two versions of *Person* are inspected.

Person	*Person*
name: "Fred Bloggs"	*surname: "Bloggs"*

To convert *"Fred Bloggs"* into *"Bloggs"* requires more information about the meaning of the data than is present in a normal class definition. The author's approach adds this missing information in the form of special methods that are responsible for converting instances from one version of a class definition to another.

Figure 3 shows three versions of a class definition linked by update/backdate methods.

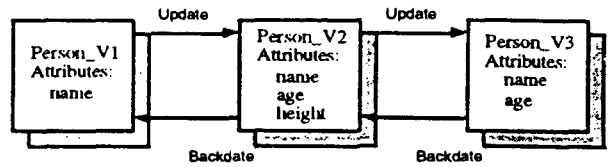

Figure 3 Versions using Backdate/Update Functions.

When a new version N of a class is defined, the backdate function for that class is defined and, at the same time, the update method for version N-1 is defined.

By following a path of update/backdate methods it is possible to convert an instance of any version of the class definition to any other version. This allows us to consider any instance to be an instance of the class as a whole rather than any one particular version of the class. The actual version to which a particular instance belongs at any one time is irrelevant as far as the end user is concerned because any query will automatically convert the instance to the version implied in the query. The version to which instances are converted will normally be the current (latest) version of the class. However to support forward compatibility (the ability for old programs to access new data) the change will sometimes be from the newer version to the older.

Since instances can be converted from the format of any version of the class to any other, it is best to consider the instances to be of indeterminate type, only being made concrete into a certain type format as the result of a query.

This is shown in figure 4. where a set of instances of indeterminate version is insulated from the outside world by a number of versions of a class definition. The instances may only be seen outside this system in one of the representations provided by the class versions. The class version used is determined by the queries attempting to extract the instances from the system.

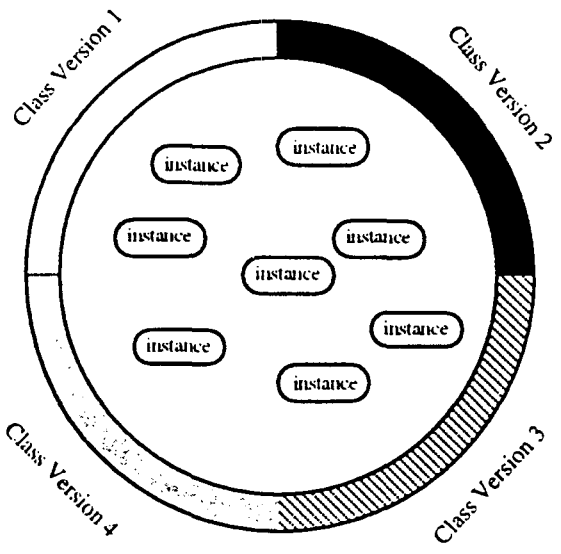

Figure 4. Dynamic Instances.

4. An Example

To illustrate how schema evolution is accomplised using CLOSQL, the following example (figure 5) is considered -

A class *Person* has been defined that has amongst others an attribute *height*. This attribute represents the height of an individual in inches. Instances of *Person* have been created, amongst them one with the name Fred, with a value of *height* of 72 inches. For some reason, it is decided that the database is to be changed to represent heights in cm rather than inches. Thus a new version of *Person* is created (*Person_2*).

At the same time, an update method must be defined for the old version of *Person* (*Person_1*) to define the conversion of an instance of that version to the new version. This will allow backwards compatibility because the update method can convert the old instances, allowing them to be used by queries expecting heights to be measured in cm. The update method simply copies the value for *name* unchanged but scales the value of *height* by 2.5. The update method allows forwards compatibility (allowing programs written assuming heights are measured in inches to read instances created with heights in cm).

As well as the update method, a backdate method is also defined for the new version of *Person* (*Person_2*) to convert instances of *Person* to the old version should the need arise.

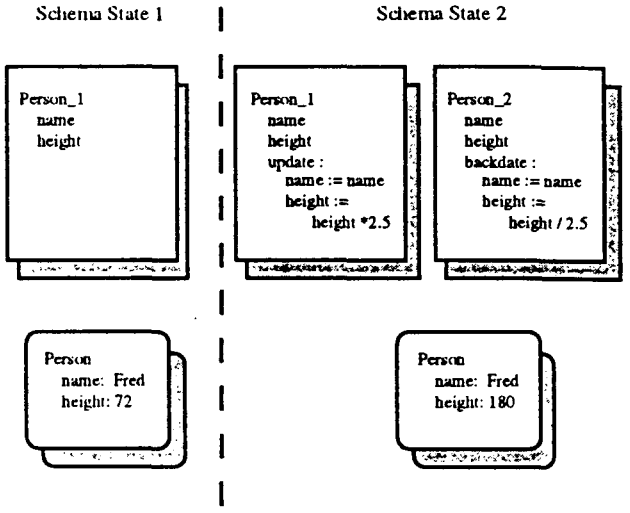

Figure 5. Changing the Units of an Attribute

A query to retrieve the value of height for the instance Fred would, by default, return the value in the format of the latest version of *Person*. ie. in cm. If however a program (or method of some other class) was expecting data in the old format, then it would use a more specific query, that requested an instance of *Person_1*. This puts the onus on the program or method author to include the version number of the class version required in any query. This can be done easily as the version number will always be known at the time of writing the code.

There is a problem in class versioning systems that results from different versions of a class representing different information. In the example of figure 6, if an instance of *Person_1*, with a value of *age* say of 27, is converted to the new version of the class definition, it loses its value for *age*. This does not become apparent until the instance is re-converted to *Person_1*, when the *age* attribute will be re-initialised to 0. CLOSQL attempts to address this problem by storing values that may otherwise be lost.

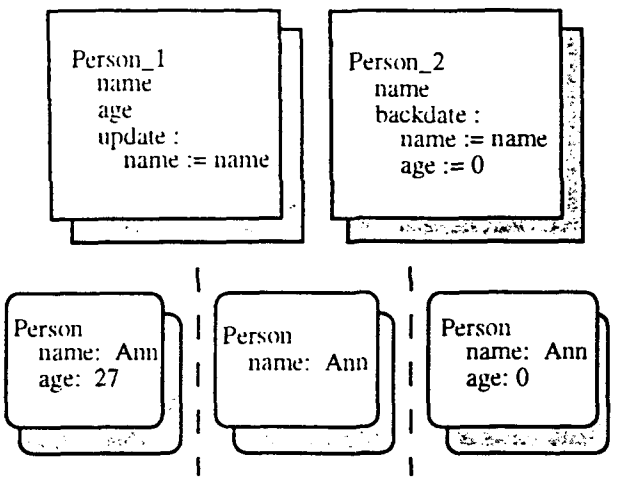

Figure 6. Lost Values as a Result of Instance Conversion.

When an instance is converted from one version of a class to another, attributes of the instance that do not have an attribute definition with the same name in the new class version are automatically stored in a system part of the database. Similarly when an instance is converted the other way, the system part of the database responsible for storing these values is checked first for a value to restore before any default value is assigned to the attribute. This is equivalent to hiding attributes that were to be removed from a class version, allowing them to re-appear when converted back to a version possessing the attributes.

While the examples in this paper have only shown dynamic instance conversion with attributes that contain primitive values, the system works equally well with values that are themselves instances of other classes. The update/backdate methods may contain code to modify instances and indeed create and delete instances.

5. Conclusions

The advantages of CLOSQL over the other systems for supporting schema evolution are :-

• Better forward and backward compatibility - Systems such as ENCORE and AVANCE that use exception handlers to achieve some degree of forward and backward compatibility suffer from the inability to assign any extra storage for new attributes. Their reliance on instance emulation rather than instance conversion means that their forward and backward compatibility is limited to providing default values for attributes that are present in one version but not in another. In CLOSQL there is no such restriction.

• Including semantics in schema changes - CLOSQL has the additional advantage that it embeds semantic information about the schema change in the database. This allows such changes as changing the units of an attribute and hence scaling the value of an attribute without necessarily changing its name. This is not possible in AVANCE and ENCORE as attributes of a given name are assumed to be identical in different versions.

The disadvantage of CLOSQL is that the end-user has to define his/her own update and backdate methods. CLOSQL has a GUI, that provides facilities for defining and representing class versions graphically, this helps to alleviate the problem by acting as a code generator, reducing the need for the user to understand any more about the syntax of these methods than is necessary. In any case, this problem also applies to the writing of any methods in an OODB.

References

[1] Banerjee, J., W. Kim, H. Kim and H. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases.", Proceedings of ACM SIGMOD, San Francisco. 1987.

[2] Björnerstedt, A. and S. Britts, "AVANCE: An Object Management System", Proceedings of OOPSLA'88, pp.206-221. 1988.

[3] Björnerstedt, A. and C. Hulten, "Version Control in an Object-Oriented Architecture", Object-Oriented Concepts, Applications and Databases, Ed. Kim, W. and F. Lochovsky, Addison-Wesley, 1989.

[4] Clamen, S. M. "Type Evolution and Instance Adaptation". Technical report No. CMU-CS-92-133. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890. 1992.

[5] Kim, W., "Introduction to Object-Oriented Databases", The MIT Press, 1990.

[6] Kim, W., J. F. Garza, N. Ballou and D. Woelk, "Architecture of the ORION Next-Generation Database System", IEEE Transactions on Knowledge and Data Engineering, 2(1), pp 109-124, 1990.

[7] Lerner, B. and A. Habermann, "Beyond Schema Evolution to Database Reorganisation", SIGPLAN Notices, 25(10), pp 67-76, 1990.

[8] Monk, S. R. and I. Sommerville, "A Model for Versioning of Classes in Object-Oriented Databases", Proceedings of BNCOD 10, Aberdeen. pp.42-58. 1992.

[9] Penney, D. J. and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", Proceedings of OOPSLA'87, pp.111-117. 1987.

[10] Roddick, J. F. "Schema Evolution in Database Systems - An Annotated Bibliography". Technical report No. CIS-92-004. School of Computer and Information Science, Faculty of Applied Science and Technology, University of South Australia, SA 5095. 1992.

[11] Roddick, J. F., "SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution", 21(3), pp 10-16, 1992.

[12] Rose, E. and A. Segev, "TOODM - A Temporal Object-Oriented Data Model With Temporal Constraints", Proceedings of Entity Relational Approach - 10th International Conference, 1991.

[13] Sjøberg, D. "Measuring Schema Evolution". Technical report No. FIDE/92/36. FIDE, Dept. Computing Science, University of Glasgow, Glasgow, G12 8QQ. 1992.

[14] Skarra, A. H. and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database.", Proceedings of OOPSLA'86, pp.483-495. 1986.