Implementing Lazy Database Updates for an Object Database System

Fabrizio Ferrandina Thorsten Meyer Roberto Zicari

J. W. Goethe-Universität
Fachbereich Informatik
Robert Mayer Straße 11-15
D-60054 Frankfurt am Main, Germany
{ferrandi,thorsten,zicari}@dbis.informatik.uni-frankfurt.de

Abstract

Current object database management systems support user-defined conversion functions to update the database once the schema has been modified. Two main strategies are possible when implementing such database conversion functions: immediate or lazy database updates. In this paper, we concentrate our attention to the definition of implementation strategies for conversion functions implemented as lazy database updates.

1 Introduction

Schema evolution in an object database system (ODBS) refers to the ability to change both the schema and consequently the database. Every time a schema is modified then the database has to be updated to be brought up to a consistent state with respect to the new schema.

Some commercial ODBSs on the market allow both updating the schema and the database.

A schema can be changed normally using special primitives, see for example [19]. In few systems, as a consequence of a schema change, the database is also modified using user-defined *conversion functions* [4], [12] which take as input parameters the old and new schema class definitions and when executed transform the objects of the database to conform to the new schema.

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

The body of a conversion function defines the desired database transformation, and it is normally defined using the data manipulation language offered by the ODBS, e.g. C++ [4], [12].

The designer has to define a conversion function for each modified class in the new schema. System default transformations are applied in case no explicit conversion functions are given by the designer [4], [9], [12], [18].

What is important for the application designer is that after definition of the appropriate conversion functions, the entire database must be in a consistent state with respect to the new schema.

From an implementation view point, conversion functions are updates to the database. There are mainly two strategies for implementing database conversion functions: immediate and lazy[1] [8].

In the first case, all objects of the database are updated immediately after the definition of the conversion functions. In the second case, objects are updated only when used (i.e. conversion functions are executed only when objects are effectively accessed by applications).

Both implementation strategies have pros and cons. In particular, supporting immediate database updates may cause overall performance degradation if class extensions are not maintained by the database system. For some systems the lazy update implementation is a preferable strategy. However, this implementation decision should not be of concern to the schema designer and ultimately to the end user.

This paper concentrates on how to implement user-defined conversion functions using lazy database updates.

We will point out a number of problems that need to be solved in order to provide a "correct" implementation of conversion functions as lazy database updates and show a way to solve them. In particular our definition of correctness will require that execution of a conversion function

[1]Lazy updates are also called *deferred* updates.

implemented as a lazy database update be equivalent to the execution of the same conversion function as if it were implemented with an immediate database update.

The rest of the paper is organized as follows: in Section 2 we will present by means of an example how conversion functions are defined and associated to one or more schema transformations. Conversion functions enable the schema designer to instruct the database system how to change the objects in the database.

In Section 3 we briefly compare the two implementation strategies to implement conversion functions: immediate and lazy. In Section 4 we point out that, when implementing conversion functions as database updates using a lazy approach, special care has to be paid to conversion functions which use objects of different classes. We call such conversion functions complex. In Section 5 we present our approach in implementing conversion function as lazy database updates. We will show the data structures and present three algorithms in details that we use for implementing such data base transformations. We will also show how immediate updates can be implemented using the basic algorithms for lazy database updates. In Section 6 and 7 respectively we will review relevant related work and present the conclusions.

2 User-defined Conversion Functions

We want to show in this section how conversion functions are normally defined and associated to a modified class. For more details the reader is referred to the user manuals of database systems such as for example ObjectStore [12], GemStone [4].

Changing the schema in an ODBS implies changing the database, i.e. the changes to the schema have to be "propagated" to all objects in the database. We will refer to this activity as database transformation. The database system should perform the database transformation after a schema modification has been made. Transformation of a database object means changing its internal structure and its value. Changing the object internal format is a task of the database system and it is done automatically without need of manual assistance by the schema designer.

Changing the object value is mainly an application-dependent task. The solution is for the schema designer or application builder to write a conversion function which tells the database systems how to change the value of the objects, and then to associate such conversion function to the appropriate class that has been changed.

If no conversion function is provided for a modified class, the system associates a default conversion function to each modified class in the schema.

In the rest of the paper we will assume applications that always work with the most recent schema definition, i.e. no notion of class or schema versioning is used.

We will use throughout the paper the O_2 Object Database System when giving examples [1], [10]. However, the results presented here are general and applicable to a broader class of ODBSs.

Let us consider at time t_0 we have defined the following *Company_schema*:

```
time t0:
class Employee type tuple ( name: string,
                            monthly_salary: real,
                            company: Company )
end;
class Company type tuple ( name: string,
                           n_employees: integer,
                           employees : set(Employee) )
end;
```

If the designer wants to change this schema, say at time t_1, and modify attribute *monthly_salary* of class Employee from *monthly_salary* to *yearly_ salary*, he/she will have to write a conversion function that tells the system how to compute the value of the *yearly_salary* and how to change all salary values of objects of class Employee already existing in the database.

This is exactly what is done by the `mod_salary` conversion function associated to the updated class Employee:

```
time t1:
modify class Employee type tuple ( name: string,
                                   yearly_salary: real,
                                   company: Company)
with conversion function mod_salary {
                      new->yearly_salary=old->monthly_salary*12;
}
end;
```

In the conversion function "->" returns the attribute value of an object. `Old` refers to an object conforming the format of class Employee before the modification has been performed, whereas `new` refers to the same object after its modification. In the conversion function the transformation is not defined for all the attributes of class Employee but only for the attribute *yearly_salary*. This is because we assume default conversion functions are *always executed on objects before a user-defined conversion function is executed.* This simplifies the writing of user-defined conversion function. In fact, in the example, there is no the need to write trivial transformations such as:

```
new->name = old->name,
new->company = old->company.
```

These transformations are taken into account by default conversions[2].

Note that a user-defined conversion function is not a property which is "inherited" in subclasses. For each sub-

[2]For reasons of brevity we do not show here the default transformation rules. Default conversion functions are automatically associated by the system to all modified classes in the schema. Default transformation rules transform existing objects to conform to the new class definition preserving, whenever possible, the old information.

class of a modified class, an explicit conversion function has to be defined or the default transformation will be used.

3 Immediate vs. Lazy Database Updates

As we mentioned before, two implementation strategies can be used to implement conversion functions:

1. Conversion Functions as Immediate Database Updates: with this approach the database system executes the conversion functions on *all* objects of the modified classes as soon as the modification on the schema is performed. All objects of the database are transformed *logically* and *physically* to be consistent with the new schema definition before any program can use again the database.

The advantage of this solution is that after execution of the conversion functions the entire database is in a consistent state wrt the new schema.

The problems with this approach are:

- All running programs have to be suspended until the application which updates the database is finished. The time when the database is locked can be very long depending on specific parameters (e.g. size of the database, type of update performed, object retrieving strategy, etc.).
- All objects of the modified classes have to be updated at once. This could be expensive, especially if the system does not manage class extensions.

2. Conversion Functions as Lazy Database Updates: with this approach objects are expected to *logically* conform to the new schema after the change to the schema has been performed, but they are *physically* restructured and transformed by conversion functions only when they are used. *Every time and only when an object is accessed and it is in an old format (and value), then it is transformed to the new definition (and value) consistent with the current definition of the schema.* Note that the schema may undergo several changes before an object is used. No database transformation is done if objects are not used, and most important only the objects that are effectively used are transformed, and not all objects of the modified classes as in the case of immediate updates.

The advantage of this solution is that we do not have to lock the entire database to execute a conversion function.

The problems with this approach are:

- There is the need to store and "remember" the *history* of all schema updates that have been performed in the system.
- Every time an object is accessed by an application, a test has to be done in order to check whether the object is already in the new format/value or not, i.e. if the object has to be updated or not.

We believe it is important that whatever strategy is chosen to implement conversion functions by the database system, there should be no difference for the schema designer as far as the result of the execution of the conversion functions is concerned [7].

This leads to our notion of correctness of a conversion function implementation. A correct implementation of a conversion function satisfies the following criteria:
Equivalence criteria:

The result of a conversion function implemented by a lazy database transformation has to be the same as if the same conversion function were implemented by an immediate database transformation.

4 Implementation Problems of Lazy Database Updates

From now on we consider an implementation of a conversion function correct if it satisfies the equivalence criteria defined in Section 3. Special care must be taken when implementing user-defined conversion functions with lazy database updates.

In subsections 4.2 and 4.3 we show two important problems that have to be taken into account when implementing conversion functions under the above assumption. First we give some basic definitions.

4.1 Complex and Simple Conversion Functions

Let us reconsider the definition of the schema *Company_schema* after its last modification at time t_1 (see Section 2).

Suppose at a later time, say t_2, we have added the attribute *tot_emp_salaries* to class Company:

```
time t2:
modify class Company type tuple ( name: string,
                                  n_employees: integer,
                                  employees : set(Employee),
                                  tot_emp_salaries: real )
with conversion function add_tot_salary {
                           emp: Employee;
                           total: real;
                           total = 0;
                           foreach (emp in old->employees) {
                                  total = total + emp->yearly_salary;  }
                          new->tot_emp_salaries = total;   }
end;
```

The attributes value is supposed to represent the total amount of money paid by the company to all its employees. This is expressed by the conversion function **add_tot_salary** associated to the class Company.

Assume later, at time t_3, we modify the type of the attribute *company* of class Employee to be a tuple instead of a reference to another class:

```
time t3:
modify class Employee type tuple (  name: string,
                                    yearly_salary: real,
                                    company: tuple ( name : string,
                                                     n_employees: integer,
                                                     tot_emp_salaries: real) )
with conversion function mod_salary {
     new->company.name=old->company->name;
     new->company.n_employees = old->company->n_employees;
     new->company.tot_emp_salaries = old->company->tot_emp_salaries;   }
end;
```

Let us now assume we make a final modification at time t_4 deleting the attribute *n_employees* from the class Company:

```
time t4:
modify class Company type tuple ( name: string,
                                  employees : set(Employee),
                                  tot_emp_salaries: real )
end;
```

The modification performed at time t_4 does not have any conversion function associated to it. This means that the default conversion is used for the transformation of the objects.

We should note in the example the difference between the conversion function mod_salary associated to Employee at time t_1 and the conversion function mod_company defined at time t_3 . For the first one the value of a "restructured" object of class Employee is computed using only the object's local value. The second one instead uses the value of an object belonging to another class in the schema, i.e. class Company.

This is an important distinction when implementing conversion functions, as we will see in the rest of this section. We will then classify conversion functions as follows:

- *Simple conversion functions*, where the object transformation is done using only the local information of the object being accessed (see the conversion function defined at time t_1).
- *Complex conversion functions*, where the object transformation is done using objects of the database other than the current object being accessed (see the conversion functions defined at time t_2 and t_3).

4.2 Correctness

Implementing *complex* conversion functions requires special care, as it will be shown in this section.

Let objects *e1* and *c1* of class Employee and Company respectively be conformed to the class definitions of the schema as defined at time t_2 (see Figure 1).

Recalling our definition of correct implementation of a conversion function (see Section 3), the expected result of a lazy transformation of *e1* and *c1* should be equivalent to the result obtained with an immediate database transformation.

In the example, if the immediate transformation were used, this would require that at time t_2 all objects of class

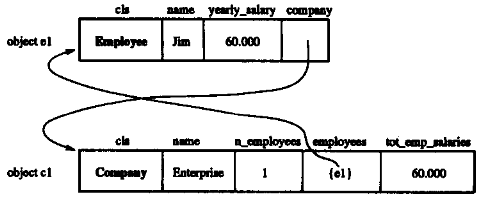

Figure 1: Object *e1* and *c1* conforming to their class definition at time t_2.

Company must be transformed before the modification on Employee at time t_3 can take place. The same is true at time t_3: all objects of class Employee must conform to the new definition of the class before the update on Company at time t_4 can be performed[3]. In Figure 2 the modifications to objects *e1* and *c1* respectively at time t_3 and t_4 are shown.

Therefore, any correct implementation of conversion functions using lazy updates in the example has to result exactly in same values for objects *e1* and *c1* defined at time t_4 as shown in Figure 2.

Suppose that at time t_4 objects *e1* and *c1* have not been accessed by an application since time t_2. Since we use lazy updates, their structure is not changed from the one they had at time t_2. If at time t_a, with $t_4 < t_a$, an application accesses object *c1*, then *c1* will be restructured by the system and its new value is computed by applying the default transformation. In Figure 3 the restructured object *c1* is shown at time t_a.

If, at time t_b, with $t_a < t_b$, object *e1* is accessed, *e1* will be restructured as well and its new value will be computed by applying mod_company defined at time t_3 (see Figure 3).

The bad news is that mod_company accesses object *c1* via the attribute *n_employees*. But *c1* now does not have anymore all the information required for the transformation of *e1* because it lost the attribute *n_employees* when it was transformed at time t_a. In this special case, the execution of mod_company would result in a run-time type error. In general, using default values for the restructured object *e1* does not solve the problem, as it could result in an incorrect database transformation.

An additional case when the equivalence criteria is not satisfied arises in the example when attribute *n_employees* is not deleted from class Company, but its value in object *c1* is changed at time t_a before object *e1* is transformed at time t_b. In the immediate database transformation the value read from *c1* would have been the one of *c1* at time t_3 and not the one at time t_a. In this case we say that

[3]To keep the exposition simpler, we suppose that an immediate database update is launched every time a modification on a class has been performed. I.e. at time t_i, all objects of the updated class are transformed before the modification at time t_{i+1} is considered. The results we present can be generalized to the case of an immediate transformation launched after more than one class transformation is performed.

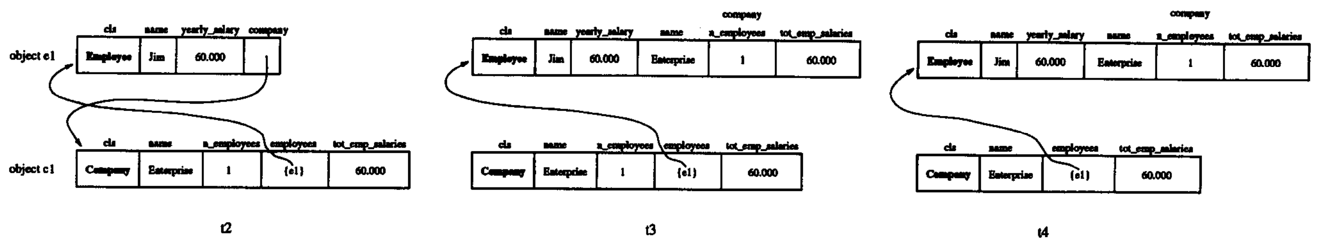

Figure 2: Evolution of objects *c1* and *e1* using the immediate database transformation.

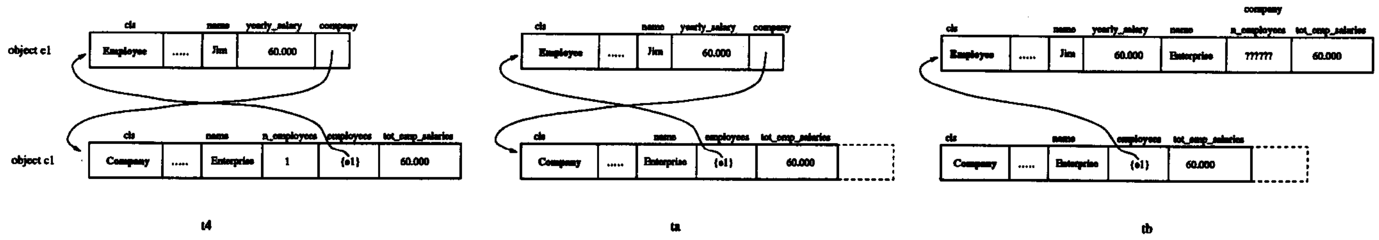

Figure 3: Evolution of objects *c1* and *e1* using the lazy database transformation.

the lazy and the immediate database transformations are not *time-equivalent.*

To summarize, the example shows that complex conversion functions cannot be mapped into an arbitrary sequence of lazy database updates to avoid generating incorrect database transformations.

In Section 5 we will present two algorithms for implementing simple and complex conversion functions using lazy database updates, which guarantee correct database transformations. In the same section we will also outline a third possible alternative implementation, known as screening.

4.3 Cycles

The second problem that need to be solved in the implementation of conversion function with lazy database updates is that of *cycles* .

Let us reconsider in our example the schema modifications performed at time t_2 and t_3. Both class modifications are associated to a complex conversion function. Note that `add_tot_salary` defined at time t_2 (associated to class Company) "uses" information belonging to objects of class Employee, and `mod_company` defined at time t_3 (associated to class Employee) "uses" information belonging to objects of class Company. This situation leads to a *cycle* in the history of conversion functions and, as we will see, to some problems during objects update.

Suppose, at time $t_c > t_4$ an application accesses an object *c2* of class Company which is in a format conforming to the definition of the class at time t_1 (see Figure 4). The conversion function `add_tot_salary` is applied in order to convert the object to the subsequent definition along the class history. Suppose now that the function accesses the object *e2* of class Employee which is in a format conforming to the definition of Employee at time t_1 (see Figure 4). The object *e2* will be transformed as well according to `mod_company`. Unfortunately, `mod_company` accesses the same *c2* under transformation. This means that, if no contra-measures are taken into account, the system would try to transform *c2* again because its format does not conform to the most recent class definition.

In order to avoid infinite loops, this cycle has to be recognized and the system should not try to transform *c2* a second time.

After detection of the cycle, the transformation of *e2* should continue. The conversion function `mod_company` could use the information present in the old *c2*. This solution does *not* always work properly. In fact, in the example, `mod_company` needs the field *tot_emp_salaries* of object *c2*, but the conversion of *c2* has been blocked before the system were able to update the field *tot_emp_salaries*. Therefore the cycle cannot be solved using the old information of *c2*. In the rest of the paper we call these cycles "*critical*".

We will show some possible implementation strategies that recognize cycles and prevent them to be *critical* in Section 5.

5 Implementing Lazy Database Updates

5.1 Data Structure

In this section we present the data structure we will use for the implementation of lazy database updates.

We assume that the physical format of an object, i.e., as it is internally stored, contains two parts: *the object header* and *the object value.*

The object value part is used for storing values that reside within the object, such as attribute values. The object header contains, among other info, the *identifier of the object's class descriptor* (`cls`) and the *type entry identifier* (`tid`) according to which format this object itself is

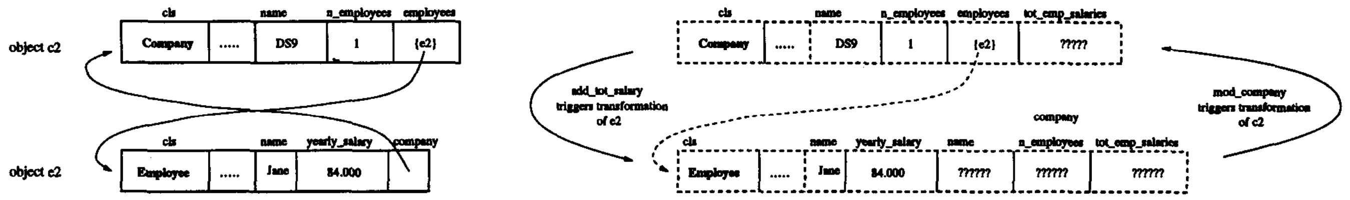

Figure 4: Cycle during objects' transformation.

stored. Each of these two can be viewed as somewhat special fields in the physical format of the object. This is illustrated in Figure 5. Objects as described here are supposed to be handled by an Object Manager and reside in secondary storage.

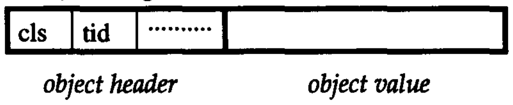

Figure 5: General physical format of an object.

We also assume that a Schema Manager maintains a set of class descriptors, each of which describes a single class in terms of its properties — attributes and methods— sub- and superclasses, *etc.*. To accommodate lazy object change, a class descriptor contains a list `tlist` of its former types. We call this list the *type history list.* An entry in `tlist` contains the following fields:

- the type `type` of this entry,
- the type entry identifier `tid`, a simple integer number, which helps in identifying to which entry an object of the class belongs,
- a field which contains a reference to a conversion function `cf` that is used to convert objects conforming to the format of this entry to the next type entry[4] in `tlist`,
- a pointer `next` for list linking purposes that identifies the entry of the prior type entry.

For the moment, we consider `tlist` to be a linked list, in which newer entries occur first: the list is in a chronologically descending order. The *current type entry* of the class (i.e. the most recent) is supposed to be kept in a similar entry as the entries mentioned above, but then in the class descriptor proper, as is illustrated in Figure 6.

The *nth* entry of `tlist` can be accessed using the array notation `tlist[n]`, whereby `tlist[0]` refers to the current type entry.

A simple integer variable called *schema_state* is associated to the schema. The *schema_state* is incremented every time a schema change modifies the structure of a class and therefore a new entry has to be created in the history.

[4]A next type entry of a class is a type entry that chronologically follows the type entry at hand.

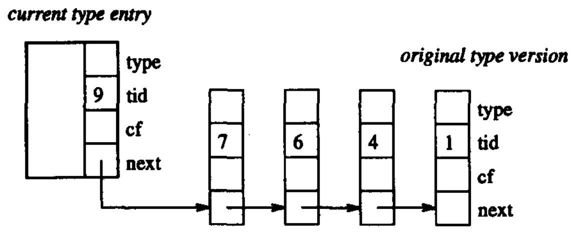

Figure 6: A class descriptor with its type history list.

Essentially, in lazy modality, schema updates only deal with the schema, and not with the objects. The objects are not changed at schema update time. In this situation, a new entry will be created at the beginning of `tlist`, and the contents of the current type entry of the class will be copied verbatim to it. The *schema_state* is incremented and it is assigned to the `tid` field of the current entry in the class descriptor. The `cf`-field information in the new entry of `tlist` points to a conversion function, if one has been provided by the designer. Otherwise the value remains *nil.*

The test for determining whether an object o is in current format will thus amount to checking whether the type entry identifier of o equals the type entry identifier of the current type of the class:

$$o\texttt{->tid} == o\texttt{->cls->tid} \tag{1}$$

The test for determining whether a user-defined conversion function has to be applied to an object o which is not in current format is:

$$o\texttt{->cls->tlist[n]->cf != } nil \tag{2}$$

whereby n corresponds to the nth entry in `tlist` where o`->cls->tlist[n]->tid == `o`->tid`. If the test returns *true*, the conversion function attached to `tlist[n]` will be applied on o and overwrites the default transformation.

To support class deletions, the Schema Manager needs to maintain an additional data structure that we call `Deleted_Classes`. This is a list of class descriptors, each in the form of Figure 6. It is the administration of formerly existing classes, including their type history lists. Whenever a class ceases to exist it is placed in this list and removed from the list of current classes. Objects of a deleted class remain in the database, therefore their value can still be used by conversion functions.

In what follows we assume that the restructuring of an object is done by the system which *preserves the Object Identifier (Oid)* of the object itself. This means that when a class definition has been changed, there is no need to change application code or other objects that reference instances of the modified class. A similar approach is used by Versant [18].

5.2 Basic Algorithms

In this section we show a basic algorithm to perform lazy database updates and how this algorithm can be used to implement immediate database updates.

The basic lazy database update algorithm is used by the system for transforming an object *o* conforming to an old class definition in the schema to its current (the most recent) definition:

```
Algorithm Lazy
while (o->tid <> o->cls->tid) do {
    create variable temp; /* temp has the same type as the type of object o */
    copy o in temp;
    restructure o in order to conform to the next
            definition of the class history;
    apply the default conversion on o;
            /* the default transformation takes the value in
            temp and performes the transformation on o; */
    find the nth entry in tlist where (o->cls->tlist[n]->tid == o->tid)
    if (o->cls->tlist[n]->cf != nil) {
        /* if a conversion function has been provided for the transformation */
        apply the user-defined cf on o;
    }
    o->tid = o->cls->tlist[n-1]->tid;
    /* update the tid of o looking at the chronologically subsequent entry in tlist */
    free temp;
}
```

We want to show now how it is conceptually possible to implement immediate lazy database updates by using the basic lazy algorithm:

```
Algorithm Immediate
foreach object o in X's extension do {}
/* this statement accesses each object of class X; the body of the foreach is empty
  because the only purpose here is to access the objects without using them */
delete the history of class X;
```

In the implementation of the immediate transformation of objects of a class using the lazy approach, the system has to access each object in the class extension. Every time an object is accessed, the system launches the lazy algorithm for the transformation. After the system has accessed the last object of the extension, all the objects of the class conform to the current definition.

If extensions of classes are not provided, alternative algorithms have to be used, according to the persistent model of the system. In O_2 , for instance, the alternative approach could be to access all objects of the database starting from the root of persistence and following the links connecting one object to another. Only when an object of the modified class is accessed, the **Algorithm Lazy** is then executed.

5.3 Correctness of Execution of Simple Conversion Functions

The example presented in Section 4 points out an interesting implementation problem: how to define a "*correct*" execution of conversion functions when implemented as lazy database updates. We should not forget, that the problems described in Section 4 refer to the implementation of complex conversion functions. We give a first result, as expressed by the following theorem:

Theorem 1 *The result of the execution of simple conversion functions implemented as a lazy database transformation (basic algorithm of Section 5.2) is equivalent to executing the same conversion functions implemented as an immediate database transformation.*

Proof sketch The proof uses the induction principle over the numbers of schema updates that are performed before accessing an object.

One schema update Let *ext(X)* be the extension of the class X and $x_i \in ext(X)$ be an instance of X. Let su_1 be an update on $X \rightarrow X_{new}$ at time t_1 with conversion function cf. In the immediate transformation, $\forall x_i \in ext(X)$, cf is applied at time t_1. In the lazy transformation, cf is applied to x_i only when accessed for the first time, i.e. at a time $t_{x_i} > t_1$. $\forall x_i \in ext(X)$, the value of x_i at time t_1 and t_{x_i} is the same, therefore the value of x_i after execution of cf at time t_1 and t_{x_i} is the same. If the value of x_i were not the same at time t_1 and t_{x_i}, x_i would have been accessed at a time t_2 with $t_1 < t_2 < t_{x_i}$, but this in in contrast with the assumption that the first access to x_i is at time t_{x_i}.

Multiple schema updates Let $su_1, \ldots, su_n$ be a sequence of updates on X done at time $t_1 < \ldots < t_n$. Assume the first access on x_i be at time $t_{x_i} > t_n$. If the equivalence is proven for n updates between time t_1 and t_{x_i}, then the equivalence is proven also in case of an additional update su_{n+1} at time t_{n+1} ($t_n < t_{n+1} < t_{x_i}$). Because of the assumption, $\forall x_i \in ext(X)$, the value of x_i after a lazy execution of $cf_1, \ldots, cf_n$ at time t_{x_i} correspond to the one obtained after an immediate execution of $cf_1, \ldots, cf_n$ at time $t_1, \ldots, t_n$. The value of x_i before cf_{n+1} is executed is the same both in immediate and lazy, therefore the value of $X_{i_{new}}$ after execution of cf_{n+1} is also the same. □

As a direct consequence of **Theorem 1** a simple solution for ensuring a "correct" lazy execution of conversion functions is to allow only simple conversion functions. With this limitation, the problems encountered with the implementation of the conversion functions described in Section 4 do not apply. This approach is used by some commercial systems, such as Itasca [9] and Versant [18].

We believe this solution is too restrictive to be used in practice. Instead, we concentrate in the next subsections in defining some implementation strategies to allow a correct implementation of *complex* conversion functions.

5.4 The Pessimistic Mix-in Database Transformation

The first algorithm is called **Pessimistic Mix-in** and works as follows: after a *complex* conversion function is associated to an updated class, an *immediate* transformation is always launched on the specified class to implement the database transformation. On the contrary, for database objects for which a *simple* conversion function has to be executed, the transformation is *deferred* until they are effectively used. We can assume the system to perform automatically the "switch" from lazy to immediate and back to lazy when necessary. The switch process should be transparent for the schema designer. The drawback of this approach is that an immediate transformation of objects is launched even when not needed.

Algorithm Pessimistic Mix-in

```
if the schema modification on a class X uses a complex cf {
    apply Algorithm Immediate on class X;
    foreach subclass Y of X do {
        if Y has changed the structure{
            apply Algorithm Pessimistic Mix-In on class Y; } } }
```

This approach avoids storing a complex conversion function in a class history. In this way, by **Theorem 1**, the execution of conversion functions is always correct. Moreover, since no complex conversion functions are present in the history of classes, no cycle will occur during the update of the objects and time-equivalence is preserved.

5.5 The Optimistic Mix-in Database Transformation

We show now an alternative implementation strategy to implement complex conversion functions, that we call *optimistic mix-in transformation*. The basic idea behind this approach is to avoid as much as possible the use of the immediate database transformations. In contrast to the pessimistic mix-in approach, the system does not launch an immediate transformation every time a complex conversion function is associated to a class, but only before executing those schema modifications which would compromise the equivalence with the immediate database transformation or lead to critical cycles.

The optimistic mix-in algorithm is useful for the class of systems for which an immediate transformation of objects in the database is a too heavy operation and need to be avoided whenever possible. It is also preferable to the pessimistic mix-in database transformation when delaying a running applications is a major performance factor. The price to pay for having an optimistic mix-in database transformation consists in managing additional data structures what we call a *dependency graph*.

Definition: The **dependency graph** G is a tuple (V, E), extended by a labeling function $l : (V \times V) \rightarrow A$. V is a set of class–vertices, one for each class in the schema. E is a set of directed edges (v, w) $v, w \in V$. A is a set of attribute names. An edge (v, w) indicates that there exists at least one complex conversion function associated to class w which uses the value of objects of class v. The function *l(v,w)* returns the names of the attributes of class v used by conversion functions associated to class w.

Evolution of the schema implies changing the dependency graph associated to the schema. By looking at the dependency graph it is possible to identify when an immediate update has to be performed due to a deletion of an attribute used by previously defined conversion functions. The use of the graph is shown with our usual example, the *Company_schema*.

In Figure 7 the evolution of the dependency graph for the schema *Company_schema* from time t_0 till time t_3 is illustrated. The conversion function defined at time t_1 uses only local defined attributes, therefore no edge appears in the graph. At time t_2 and t_3, the edges are added to the graph because of the definition of the complex conversion functions **add_tot_salary** and **mod_company**.

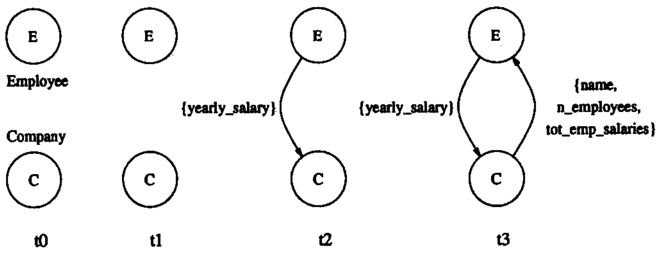

Figure 7: Evolution of the dependency graph of schema *Company_schema*

The dependency graph has to be checked by the system every time an update to the schema modifies the structure of a class. If a schema update results in an attempt to delete attributes needed by conversion functions previously defined, then it is implemented with an immediate database transformation on the appropriate classes before actually updating the schema.

At time t_4, when the attribute *n_employees* is deleted from the class Company, the system would detect an edge in the dependency graph labeled with *n_employees*. Before performing the deletion of the attribute in the class Company, the immediate database update is launched on the extension of class Employee.

Note that in contrast to the pessimistic mix-in approach, the immediate database transformation is used

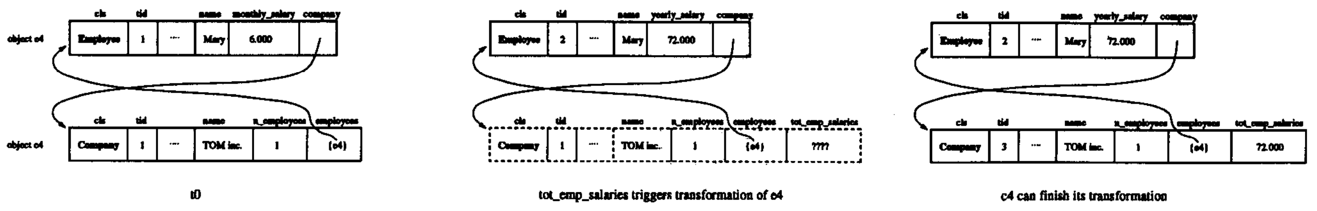

Figure 8: Objects *e4* and *c4*.

only when deletion or modification of information could compromise the correctness of lazy execution of conversion functions.

However, this solution does not prevent cycles to appear in the dependency graph (see the cycle at time t_3 in Figure 7). The presence of a cycle, as already pointed out in Section 4.3, could block in an irreversible way the evolution of the objects in the database. Moreover, even if a solution is found to avoid infinite loops, a strategy has to be used which treats the situation of what we called *critical* cycles in Section 4.3.

The approach has to be changed as follows:

- To detect cycles when updating the database, a *mark* is set on each object *o* which is under transformation. The mark can be implemented as a bit flag in the header of the object. When an object which is marked is accessed a second time, the system recognizes the presence of a cycle and no conversion is performed on the object.

- To avoid critical cycles we *stop the transformation of the objects before having reached the current state.* Only the object *o* which is accessed by an application will be transformed up to the current definition in the schema. Objects which are accessed by a conversion function `cf` used for transforming *o* are converted up to the entry in the history of the class which corresponds to the class definition "visible" by `cf` at the time when it was defined. The concept of visibility is modeled by the *schema_state* of the schema and, as a consequence, by the `tid`'s attached to each entry in the type history list of a class.

Suppose `cf` performs a transformation from an entry in `tlist` with `tid = i` to an entry with `tid = j`. The *schema_state* when `cf` has been defined was j-1 (see Section 5.1). The type of the other classes cl_i in the schema visible by `cf` is the one found in their nth `tlist` entry where:

$$cl_i\texttt{->tlist[n]->tid < j}$$

and the chronologically subsequent entry (if any)

$$cl_i\texttt{->tlist[n-1]->tid >= j}$$

The state of the schema when a conversion function is defined makes the system understand up to which entry an object accessed by a conversion function has to be transformed[5].

[5]It might be the case that objects accessed by a `cf` have a `tid` $\geq j$. In this case no transformation is triggered on them because they are already containing the information needed by `cf`.

Figure 8 shows the initial format of objects *e4* and *c4* conforming to the definition of the schema *Company_schema* at time t_0.

When object *c4* is accessed at time t_c, $t_3 < t_c < t_4$, the conversion function **add_tot_salary** transforms *c4* to conform to the next entry in the class history, i.e. the one with `tid = 3`. **Add_tot_salary** accesses *e4* whose `tid = 1`. When accessed, object *e4* is transformed by **mod_salary** to conform to the history entry with `tid = 2`. After the transformation *e4* is in the format needed by the conversion function **add_tot_salary** because its `tid` is < 3 whereas the one in the subsequent entry in the history (`tid = 4`) is > 3. Therefore, *e4* can be used as it is and its conversion is stopped. Since the conversion function **mod_company** is not executed, no critical cycles occur. **Add_tot_salary** can continue the transformation of *c4* up to the current entry in `tlist`. If, subsequently, *e4* were accessed by an application, the appropriate information for the update would be found in *c4*.

To support a lazy transformation which may be stopped before the object has reached the current format, the basic **Algorithm Lazy** presented in Section 5.2 has to be modified into what we call the **Algorithm Blockable Lazy**:

Algorithm Blockable Lazy (up_to_tid: integer)

```
while (o->tid < up_to_tid and o not marked) do {
    /* if o is not already involved in a transformation, then convert it to conform
       to the enrty in tlist whose tid = up_to_tid */
    create variable temp; /* temp has the same type as the type of object o */
    copy o in temp;
    restructure o in order to conform to the next
            definition of the class history;
    mark o;   /* useful for avoiding infinite loops */
    apply the default conversion on o;
    find the nth entry in tlist where (o->cls->tlist[n]->tid == o->tid);
    if (o->cls->tlist[n]->cf != nil) {
        apply the user-defined cf on o;
    }
    o->tid = o->cls->tlist[n-1]->tid;
    unmark o;
    free temp;
}
```

As for the **Algorithm Lazy**, the **Algorithm Blockable Lazy** is executed by the system every time an ob-

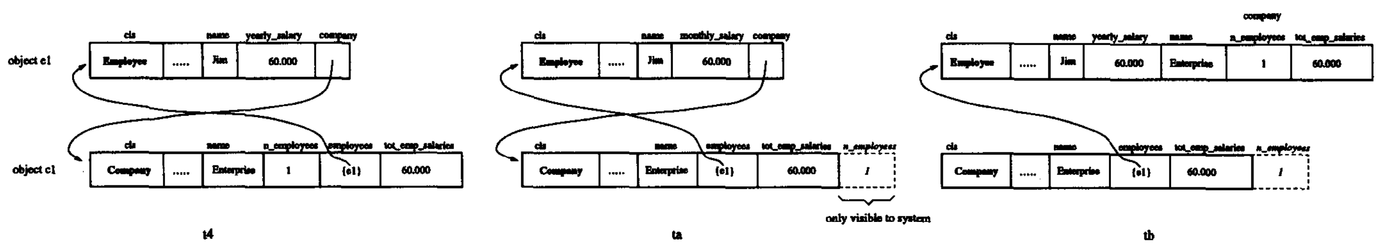

Figure 9: Evolution of objects *c1* and *e1* using the screening approach.

ject is accessed in the database. This modified algorithm differs from the basic **Algorithm Lazy** (Section 5.2) because of the input parameter **up_to_tid** indicating the entry in the class history up to which an object has to be transformed. The optimistic mix-in database transformation makes use of the **Algorithm Blockable Lazy** when objects are accessed. We assume that the system can distinguish between the cases where an object has to be transformed up to the current entry in the class history, from the case where the transformation has to be stopped up to a generic entry in the history of a class [6].

The definition of algorithm **Optimistic Mix-In** is given in the next box. The algorithm is executed by the system every time the schema is modified by the designer.

Algorithm Optimistic Mix-in

```
switch depending on the type of schema change:
  create class X :  V = V + {X};
                    /* a new vertex is added to the dependency graph */
  delete class X :  /* the vertex X is not deleted from the dependency graph
                       because objects of X might be accessed by conversion functions */
  modify class X : switch depending on the kind of class update:
      delete attribute att_of_X :
          /* all classes using att_of_X in a cf have to be updated */
          for each Y where (X,Y) in E  and  att_of_X in l(X,Y)  do {
              apply Algorithm Immediate on class Y;
              remove all edges (v,w) from E where w=Y;
              /* restructure the graph according to the
                 immediate transformation on class Y */
          }
      add attribute att_of_X :
          foreach complex reference to an attribute att_of_Y
          of class Y in a complex cf  do {
              add a new edge (Y,X) to E if it does not exist;
              add {att_of_Y} to the result of the labeling function l(X,Y);
          }
      modify type of attribute att_of_X :
          /* for the graph, we consider this operation as a deletion followed
             by an addition of the attribute */
          for each Y where (X,Y) in E  and  att_of_X in l(X,Y)  do {
              apply Algorithm Immediate on class Y;
              remove all edges (v,w) from E where w=Y;
              /* restructure the graph according to the
                 immediate transformation on class Y */
          }
          foreach complex reference to an attribute att_of_Y
          of class Y in a complex cf  do {
              add a new edge (Y,X) to E if it does not exist;
              add {att_of_Y} to the result of the labeling function l(X,Y);
          }
```

5.6 Screening

We present now an alternative implementation strategy known with the name of *screening*. Screening resembles the pure lazy approach, i.e. no immediate transformation has to be launched for updating the database. Basically, when some information is deleted from the schema, it is only logically filtered out, but not physically deleted in the database. When, for instance, a deletion of an attribute (or a change in the type which would correspond to a deletion and an addition of the same attribute) is performed, the update is not physically executed on the object structure but simply a different representation of the object is presented to the user. With this approach, the Schema Manager is in charge of managing the different representations of the objects (one representation visible to applications and one representation visible to conversion functions).

We illustrate how this strategy works using our example. Figure 9 illustrates the example presented in Section 4.2 (where the attribute *n_employees* has been deleted at time t_a from object *c1* of class Company) using the screening approach.

Using screening at time t_b the conversion function mod_company this time can be executed because the information it needs has not been physically deleted, it is still in the object.

If no space optimization is taken into account, since information is never deleted in the objects, the size of the database risks to grow continuously. If disk space is a performance factor, then a possible optimization is to physically delete the information in those objects which will never be accessed by any complex conversion function. This can be easily obtained by checking the dependency graph presented in Section 5.5. Objects of classes which do not have any outgoing arrow in the dependency graph should not contain deleted attributes because no conversion function will ever use them.

Using a *screening* approach, critical cycles have to be handled in a similar way as described in Section 5.5.

[6]For keeping the exposition simpler, we do not show here how the system infers up to which entry an object has to be transformed.

Algorithm Screening Blockable Lazy (up_to_tid: integer)

```
while (o->tid < up_to_tid and o not marked) do {
    create variable temp;
    copy o in temp;
    restructure o in order to conform to the next definition of the class
        history preserving the information which has been deleted;
    mark o; /* useful for avoiding infinite loops */
    apply the default conversion on o;
    find the nth entry in tlist where (o->cls->tlist[n]->tid == o->tid);
    if (o->cls->tlist[n]->cf != nil) {
        apply the user-defined cf on o; }
    o->tid = o->cls->tlist[n-1]->tid;
    unmark o;
    free temp;
}
```

When an object is accessed by an application, the system executes an algorithm we call **Screening Blockable Lazy** with a `tid` parameter according to the application or the conversion function which accessed it. The algorithm **Screening Blockable Lazy** is presented at the end of this subsection.

The main difference of this new algorithm with respect to the **Blockable Lazy Algorithm** presented in Section 5.5 is that the logical deletion of the attributes is taken into account. No physical restructuring of the objects is needed when information is deleted from the schema. Space optimization is not considered in the algorithm.

Note that both the *optimistic* database transformation and *screening* do not preserve the time-equivalence with the immediate database transformation. If the designer requires time-equivalence, he/she has to explicitly launch the immediate database transformation after the schema has ben modified.

6 Related Work

Not all available ODBSs provide the feature of adapting the database after a schema modification has been performed [14], [15], [17]. For those that do it, they differ from each other in the approach followed for updating objects. Some commercial systems support the possibility to define object versions to evolve the database from one version to another such as *Objectivity* [13] and *GemStone* [4]. Objectivity does not provide any automatical tool to update the database besides the fact to provide versions. The designer has to write a program which reads the value *old_val* of objects of the old version, computes the new value *new_val* and assigns it to the correspondent objects of the new version. The program can be written in order to transform the database both immediately and lazily. GemStone, instead, provides a flexible way for updating the instances. It provides default transformation of objects and the possibility to add conversion methods to a class. Conversion methods can update objects either in groups (for instance the whole extension of a class) or individually. The transformation of the database is performed lazily but manually, i.e. objects are transformed on demand only when applications call the transformation methods. The problems pointed out in this paper do not occur when versioning is used because objects are never *transformed*, but a new version is created instead. Therefore the information for the transformation of an object can always be found in its correspondent old version.

On the other hand, the majority of the existing commercial available systems do not use versioning for updating the database. Applications can run on top of the schema as defined after the last modification. Instances are converted either immediately or lazily. *ObjectStore* [12] makes use of the immediate database transformation. So called transformation functions, which override the default transformation, can be associated to each modified class. Objects are not *physically* restructured, but a new object (conforming the definition of the modified class) is created instead. The transformation function reads the value in the old object and assigns it (after having made some modification on it) to the new object. All references to the object have to be updated in order to point to the new created object. This technique resembles the one used by those systems providing versions, the only difference being that, after the transformation, the old objects are discarded. Lazy transformation of objects is provided in systems like *Itasca* [9] and *Versant* [18]. They both do not provide the user with flexible conversion functions like the one presented in the paper. Instead, they have the possibility to override a default transformation assigning new constant values to modified or added attributes of a class.

Among prototype systems, we can mention *Avance* [3], *CLOSQL* [11], and *Encore* [16] as those ones using versioning. *Orion* [2] uses a lazy approach where deletion of attributes is filtered. Information is not physically deleted, but it is no more usable by applications. No conversion functions are provided to the schema designer.

Substantially, if we consider only those systems using a lazy database transformation, no one is currently offering to the designer conversion functions like the ones presented in this paper. Updating the database using only default transformation of objects is sometimes not flexible and powerful enough.

7 Conclusions

In this paper we discussed the lazy database transformation approach and we presented implementation problems related to it. We introduced the notion of conversion functions and showed that when complex conversion function are used, problems due to deletion of information and the presence of critical cycles can compromise the result of having a lazy database transformation which is equivalent to an immediate transformation. The equivalence can be preserved using one of the two algorithms described in this paper which make use (one more heavily than the other) of the immediate transformation. An alternative strategy also shown is to use a screening approach.

The decision which algorithm is to be used does not have a simple answer. It depends on the object persistence model associated to the particular ODBS.

We are currently evaluating the suitability of the algorithms for different ODBSs, basically concentrating on performance evaluations [6].

Acknowledgments

The work presented in this paper is partially supported by the Goodstep project funded by the Commission of the European Communities under the ESPRIT III programme.

We would like to thank Rolf de By, Guy Ferran, and Jöelle Madec for the interesting and stimulating discussions which helped us to better understand the problems.

References

[1] F. Bancilhon, C. Delobel, and P. Kanellakis eds. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[2] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of ACM SIGMOD Conf. on Management of Data*, San Francisco, CA, May 1987.

[3] Anders Bjornerstedt and Stefan Britts. Avance: an Object Management System. In *Proceedings of OOPSLA*, San Diego, CA, Sep 1988.

[4] Robert Bretl, David Maier, Allan Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone Data Management System. In Won Kim and Frederick H. Lockovsky, editors, *Object-Oriented Concepts, Databases and Applications*, chapter 12. ACM Press, 1989.

[5] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. Technical report no. 1/94, J.W. Goethe Universität, March 1994.

[6] F. Ferrandina, T. Meyer, and R. Zicari. Lazy Database Updates Algorithms: a Performance Analysis. Technical report, J.W. Goethe Universität, 1994. In preparation.

[7] F. Ferrandina and R. Zicari. Object Database Schema Evolution: are Lazy Updates always Equivalent to Immediate Updates? *Presented at the OOPSLA Workshop on Supporting the Evolution of Class Definitions*, Washington, September 1993.

[8] G. Harrus, F. Vélez, and R. Zicari. Implementing Schema Updates in an Object-Oriented Database System: a Cost Analysis. Technical report, GIP Altaïr, 1990.

[9] Itasca Systems, Inc. *Itasca Systems Technical Report Number TM-92-001. OODBMS Feature Checklist. Rev 1.1*, Dec 1993.

[10] C. Lécluse and P. Richard. The O_2 Database Programming Language. In *Proceedings of the 15th International Conference on Very Large Databases*, Amsterdam, Aug 1989.

[11] Simon Monk and Ian Sommerville. A Model for Versioning Classes in Object-Oriented Databases. In *Proceedings of the 10th British National Conference on Databases*, Aberdeen, Scotland, July 1992.

[12] Object Design Inc. *ObjectStore User Guide, chapter 9*, 1993.

[13] Objectivity Inc. *Objectivity, User Manual, Version 2.0*, Mar 1993.

[14] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and Implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[15] Bernhard Schiefer. Supporting Integration & Evolution with Object-Oriented Views. *FZI-Report 15/93*, July 1993.

[16] Andrea H. Skarra and Stanley B. Zdonik. Type Evolution in an Object-Oriented Database. In Shriver and Wegner, editors, *Research Directions in Object-Oriented Programming.*

[17] O_2 Technology. *The O_2 User Manual, Version 4.3*, July 1993.

[18] Versant Object Technology, 4500 Bohannon Drive Menlo Park, CA 94025. *Versant User Manual*, 1992.

[19] R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In *Building an Object Oriented Database System - The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.