Schema Versions in Object-Oriented Database Systems

Sven-Eric Lautemann
Fachbereich Informatik (20)
Datenbanken und Informationssysteme (DBIS)
Johann Wolfgang Goethe-Universität Frankfurt
Robert-Mayer-Str. 11-15
D-60325 Frankfurt am Main, Germany
lautemann@informatik.uni-frankfurt.de

Abstract
Object-oriented database management systems (OODBMS) are especially suited to model complex and highly dynamic application domains. In this paper, we propose a schema versioning approach which supports the dynamic change of an object-oriented database schema while it is used by running applications. Our mechanism allows to have applications working with different schema versions on top of the same single database in parallel. A flexible and parameterized approach is presented to make instances of the database accessible in different versions of a schema. In this way, it is no longer required to update all database applications at once whenever the schema is changed. Instead, the adaptation of old applications to a new schema version can be done later if this is considered advantageous.

Keywords object-oriented databases

1 Introduction
OODBMS have been developed mainly to model highly dynamic application scenarios where not only the data, but also its structure (i.e. the schema) is subject to change. As it is practically impossible to adapt all running applications whenever the schema is updated, we employ a schema versioning mechanism that keeps the original (old) schema version while only a copy can be modified to derive a new schema version.

In this way, applications that have not been adapted to a schema update can continue to work with old schema versions while in parallel other applications can work with the new schema version. The remaining problem is located at the database level: How can applications cooperate on common data if they work with different schema versions? To solve this problem, we introduce a mechanism to make instances which have been created under one schema version visible in other versions as well. This so-called propagation can be declaratively controlled by several parameters, restricting the visibility of an object to a certain subset of all schema versions. This feature is required not only during the development phase of new schema versions and applications but also later to enforce certain general constraints. Running applications can continually work with old schema versions and do not have to be adapted to every schema update.

The remainder of this paper is organized as follows: Section 2 informally introduces some elements of a basic model which are required for our approach. The main contributions of this paper are presented in Sections 3 and 4. Section 3 deals with evolution at the schema level, while Section 4 introduces a way of mapping schema changes into the database that enables one to work with overlapping parts of a single database in different versions of one schema. After an overview on related research in Section 5, we give a succinct conclusion in Section 6.

2 The Basic Model
Our schema versioning approach is based on a common object model as it is offered for instance by O2 [1, 4]. Therefore the approach can easily be adapted to various systems.

An object is an encapsulated entity which consists of a unique identifier (oid), a typed data structure which holds the current state of the object, and a behaviour which is given by a set of methods that operate on the object's state.

Each object is instantiated from a single class which defines the object's structural and behavioural properties. Classes are arranged in an inheritance DAG (directed acyclic graph) where subclasses inherit properties from their superclasses. The intension of a class defines a unique class name, a list of direct superclasses, a data type, and a set of applicable methods. The
(direct) extension of a class means the set of all objects which are (direct) instances of that class.

A schema is a set of class intensions and a (data)base is a set of class extensions.

To perform schema updates we use a taxonomy of primitives enhancing that presented in Banerjee et al. [2]. As our schema versioning approach is independent of certain primitives, the taxonomy can be easily extended without impact on our model.

3 The Schema Versioning Model

3.1 A motivating Example

To motivate our approach, we use an example out of financial business. A piece of a database schema 

\begin{verbatim}
schema sv1 {
class Company {
type tuple {
name : string,
st&xchnge : string;
method ...
} /* class Company */
class Share {
type tuple {
company : Company,
currency : string,
date : Date,
price : [0..23] int;
method ...
} /* class Share */
}
/* sv1 */
\end{verbatim}

The attribute price of a share object holds the prices at the end of each hour of a day. For the next day, a new object will be created in case this company's shares are traded. Old objects will be deleted when they are no longer required, e.g. to calculate a 30 days average price.

After some time, a new law might require that the bank keeps the complete history of the prices of all shares it trades, i.e. objects must never be deleted. In contrast to bank internal requirements to store 24 prices each day, the law only requires to store one average price for every day. Here, the following schema would be completely sufficient.

\begin{verbatim}
schema sv2 {
class Share {
type tuple {
company : string,
currency : string,
date : Date,
price : int;
method ...
} /* class Share */
}
/* sv2 */
\end{verbatim}

On January, 1 2001 national currencies in Europe will be replaced by one common currency, called Euro. Therefore, it is no longer necessary to store the currency of a share's price. Assume at the same time the bank decides to store additional information about the companies. This is expressed in a third version of the used database schema.

The requirements described result in three different versions of the bank's database schema (see Figure 1), which are used one after the other (e.g. sv1 replaces sv11) or in parallel (e.g. sv2 and sv3). Furthermore, if applications working on top of sv1 are not adapted to sv3 immediately after the derivation of sv3, even sv1 and sv3 might have to be used in parallel.

An overview on schema evolution mechanisms and a proposal for a general framework can be found in Lautemann [7].

3.2 The Schema Derivation DAG

A schema version specifies a complete schema that represents one out of several design alternatives developed in parallel or an intermediate result of a sequential development. Compared to class versioning approaches (see Monk and Summerville [9]) where the schema designer has to make sure that he is using a consistent configuration of class versions, this inter-class consistency is given automatically in the schema versioning approach.

Schema versions used by different applications are stored in a schema derivation DAG as the one shown in Figure 1. This DAG represents the is-derived-from relationship between existing schema versions. It is rooted in a system-provided schema version sv0 which also serves as a starting point for the evolution of a schema during its life cycle.

If a new schema version, svn, is necessary to meet changed requirements, any of the schema versions in the DAG can be used as a starting point for the so-called schema derivation process. After a suitable starting schema version svn is selected, the schema derivation process starts with the schema updates required to change svn into svm. Then some specifications on the object level (see Section 4) have to be done to tell the system how to handle objects which already exist or will be created later. When the schema derivation process is finally completed (considered as derivation time of svn), the new schema version svn is included into the schema derivation DAG as a child of svn. Schema version svn can then be used by applications and can serve as a starting point for further schema updates. svn is called direct subversion of svn and svn is called direct supervision of svn.

\footnote{Formerly known as European Currency Unit (ECU).}

\footnote{In the examples shown in this paper the schema derivation structure is only a tree. For examples of schema version integration leading to a DAG see Lautemann [6].}
3.3 The Schema Updates of the Example

The following statements of our schema derivation language can be used to produce the schema derivation DAG shown in Figure 1. Due to space limitations, we cannot present the complete schema derivation language here.

Schema version $sv_1$ is derived from $sv_0$ (which means it is, in fact, created from scratch) by creation of classes Company and Share.

```
derive schema version $sv_1$ from $sv_0$ {
  create class Company {
    type tuple {
      name : string,
      stock_xchange : string;
      method ...
    } /* class Company */
  } /* class Share */
}
```

Schema version $sv_3$ is derived from $sv_1$ by type modification of attributes company and price of class Share and deletion of class Company.

```
derive schema version $sv_3$ from $sv_1$ {
  modify class Share {
    delete attribute currency;
  } /* class Share */
  modify class Company {
    add attribute founders : string;
    add attribute location : Place;
  } /* class Company */
}
```

Finally $sv_3$ is derived from $sv_1$.

```
derive schema version $sv_3$ from $sv_1$ {
  modify class Share {
    delete attribute currency;
  } /* class Share */
  modify class Company {
    add attribute founders : string;
    add attribute location : Place;
  } /* class Company */
}
```

4 Mapping Schema Evolution into the Database

Our goal is to allow different applications to work with different schema versions in parallel (while each application is working with only one schema version). Because the schema versions differ from each other, the parts of the complete database which are visible in these schema versions are also different. The part of a complete database which is visible for applications working on top of a specific schema version $sv$ is called the instance access scope of this schema version ($IAS(sv)$). In the example we have $IAS(sv_0), \ldots, IAS(sv_3)$.

But we do not want to have several copies of a single object in the IASs of different schema versions of one DAG neither at the physical nor at the logical level. Instead, we offer the possibility to dynamically propagate objects between schema versions along the is-derived-from relationship. Therefore, the IAS of each schema version will be partitioned into two subsets: The first partition which is called the direct instance access scope $DIAS(sv)$ contains all objects which have been created by applications working directly on
top of sv. The second partition, called *indirect instance access scope* IIAS(sv) contains all objects which are propagated (directly or indirectly) from databases of super- or subversions of sv. Of course, we have IAS(sv) = DIAS(sv) ∪ IIAS(sv).

### 4.1 Database Conversion Functions

As already mentioned in the introduction, we want to enable different applications to cooperate on a common database, even if those applications work with different versions of the same schema. As different schema versions might well define different types for the same class, we need a way to map an object (e.g. a share) between its different representations (see Figure 2).

![Figure 2: Two different versions of a share object.](image)

To implement this mapping, we use so-called *database conversion functions* which have to be specified in the schema derivation process. Because sv3 is newer than sv1, we call the conversion function from sv1 to sv3 forward (in time) conversion function, denoted as fcfShare,3→1. Analogously, the conversion function from sv3 to sv1 is called backward conversion function, denoted as bcfShare,1→3.

Conversion functions have only to be defined between schema versions which are direct super- and subversions of each other. Propagation between arbitrary schema versions can then be done automatically by composition of conversion functions along the arcs of the schema derivation DAG (see Subsection 4.3.1).

Conversion functions are implemented at the class level, i.e. they take an object value of one class version svu.c of a source schema version svu as an input parameter and compute an object value of class version svv.c as output. Note, that svu.c and svv.c refer to versions of the same class c which are part of svu and svv, respectively. Of course, svu.c and svv.c can declare different types, like sv1.Share and sv3.Share in the example.

Conversion function fcfCompany,3→1 could for instance look as follows:

```plaintext
conversion function {
    new.name = old.name;
    new.street,change = old.street,change;
    new.founders = NULL;
    new.location = NULL;
}
```

The prefixes old and new are used to refer to the attribute values visible in the source and destination schema version (sv1 and sv3 as shown in Figure 1) and can be omitted in case of unique attribute names.

Trivial conversion functions like fcfCompany,3→1 shown above can be deduced by the system from the performed schema updates: Attributes added to a class in a new schema version (e.g. *founders* and *location* of class *Company* in sv3) are initialized with a NULL value, unmodified (e.g. attribute name) or simply renamed attributes keep the values of the original. Furthermore, obvious type conversions like from integer to real can be handled automatically as well.

The schema designer can, of course, modify those *default conversion functions*, whenever this is required to propagate the maximum object semantics. An example of a statement in our propagation language containing a user defined fcfShare,3→1 to compute a useful value for attribute price could look as follows:

```plaintext
propagate instances from sv1.Share to sv3.Share
propagation flags scmd; /* Subsec. 4.3 */
conversion function {
    new.company = old.company;
    new.date = old.date;
    for (i=0, i<24, i++)
        new.price[i] = old.price[i] * exchange_rate(old.currency); }
```

The company attribute is a reference to an object of class *Company* both in sv1.Share and in sv3.Share. If the company object is not visible in the destination schema version's database (IAS(sv3)), the reference is set to NULL during the propagation of the share object.

Note that in contrast to other conversion functions of the example, fcfShare,3→1 will read attribute values of referred company objects (including the statement new.company = old.company.name). Such conversion functions are said to be complex.

### 4.2 Object Versions

In this paper, an object (identified by its *oid*) always belongs to the same class (identified by its *cid*). This class’ type can, of course, be changed from one schema version to the next. The object can be visible in different schema versions (i.e. it can exist in several IASs) at the same time. The values of the object which are visible in these schema versions can be different, even if the object’s class did not change. In this paper the values of an object visible in different schema versions are called *object versions* (extending the common use of this phrase, where versions of one object have

---

3We assume *exchange_rate(curreny)* is an external function that returns how many Euros are equal to one unit of currency.
to have the same type). Formally, \( sv.o \) denotes the version of object \( o \) which is visible in schema version \( sv \). We say an object version is derivable from another version of the same object if it can be computed by the application of the provided conversion functions. This means that derivable object versions do not have to be stored physically (see Subsection 4.5).

### 4.3 The Propagation Flags

At schema version derivation time, the schema designer has to decide which parts of the superversion's database should be shared by the new schema version. The specification of the amount of shared data has to be done at the class level, i.e. for every conversion function it must be decided if objects of the corresponding class should be visible in the destination schema version.

When a new schema version, \( sv_u \), is derived from \( sv_u \), the schema designer may define four propagation flags to define the relationship of \( IAS(sv_u) \) and \( IAS(sv_u) \) for every propagated class.

First of all, at schema version derivation time, the schema designer might want to take a snapshot of some classes of \( IAS(sv_u) \), i.e. the system makes all instances of the selected classes of \( sv_u \) which exist at derivation time of \( sv_u \) visible in \( sv_u \) as well. For instance a share object which already existed in the extension of \( sv_1 \)'s class Share when \( sv_2 \) has been derived, will be accessible in \( sv_2 \) as well because the snapshot flag (s) was set for class Share of \( sv_u \) (see Figure 1).

In the remainder of this paper, we will use a simplified type of figure to show the semantics of the flags regarding only a single class (Employee) and the visibility and value of a single object (Smith) of this class. Of course, the mechanism works analogously for all classes and all objects (w.r.t. the flags set for their classes). In the figures, schema versions are shown as circles which are either empty, indicating that the concerned object (Smith) is not visible, or the circles are filled or crossed indicating different visible values. If we use the same symbol to show the visibility of an object in two schema versions (both circles filled or both crossed), this indicates that one of them is derivable from the other, otherwise this is not the case. Important flags are noted to be either switched on (s) or off (x). Flags we do not care about in a certain figure are not shown. For simplicity, we will first consider only forward propagation.

Figure 3 shows how the visibility of object Smith in the newly derived \( sv_2 \) depends on the snapshot flag. If the flag is switched on (s, as shown in the upper part of Figure 3), employee Smith (and all objects of classes for which the snapshot flag is set) becomes visible in \( IAS(sv_u) \) at time \( t_2 \) (schema version derivation time of \( sv_u \)). As the object value of Smith which is visible in \( sv_2 \) (denoted as \( sv_2.Smith \)) has been created by application of conversion function \( cf_{femp,2}^{1} \), \( sv_2.Smith \) is derivable from \( sv_1.Smith \) at \( t_2 \).

![Figure 3: The semantics of the snapshot flag.](image)

If the snapshot flag is not set (x, as shown in the lower part of Figure 3), employee Smith will not be visible in \( sv_2 \).

After \( sv_u \) has been derived, further changes can happen in \( sv_u \)'s database \( IAS(sv_u) \), e.g. the salary of employee Smith can be increased. The effect of such changes on \( IAS(sv_u) \) can be specified with three additional flags, namely the creation (c), modification (m), and deletion (d) flag. If the creation (modification, deletion) flag is switched on, further creations (modifications, deletions) in \( sv_u \) will be propagated to \( sv_u \)'s database \( IAS(sv_u) \). Therefore, every object which will be created in \( sv_u \) will also appear in \( sv_u \) (see Figure 4).

![Figure 4: The semantics of the creation flag.](image)

Note the difference between the snapshot and the creation flag: while the snapshot flag only considers objects which already exist at schema version derivation time, the creation flag determines the visibility of objects which will be created in the superversion after the new schema version has already been derived.

Similarly, object modifications and deletions in \( sv_u \) can be propagated to \( sv_u \). Of course, a modification or deletion of an object \( o \) in \( sv_u \) can only be propagated to \( sv_u \) if \( o \) is visible also in \( sv_u \), i.e. \( o \) must have been propagated (either by the snapshot or the creation flag) earlier. If the modification
Figure 5: The semantics of the modification flag.

If the m-flag (d-flag) is switched off, $IAS(sua)$ will not be affected by modifications (deletions) of applications on top of $sv_1$. Figure 6 shows the propagation of deletions.

Figure 6: The semantics of the deletion flag.

During the process of deriving schema version $sv_6$ from $sv_n$, the schema designer has to specify which kind of propagation is required. For each required forward propagation from $sv_n$ to $sv_6$, he has to provide one instance propagation statement (like the example given in Subsection 4.1) must be provided. In Figure 1 the flags for our example are included beside the arrows representing the conversion functions.

For backward propagation the snapshot flag does not make sense because $DIAS(sv_o) = \emptyset$ at derivation time of $sv_o$.

4.3.1 Transitive Propagation

As we already mentioned, instances can be propagated not only from one schema version to its direct super- and subversions, but also transitively to indirectly related versions. If for instance employee Smith is created in $sv_1$ after $sv_2$ and $sv_3$ have been derived (both having the creation flag set), then object Smith will also be created in $IAS(sv_2)$. This is then regarded as a creation under $sv_2$ which in turn triggers the propagation to $sv_3$ (see Figure 7).

In this way, creations, modifications, and deletions of an object in a schema version $sv_o$ might be propagated to the databases of a large set of schema versions. This set contains exactly the biggest subgraph of the schema derivation DAG rooted in $sv_n$ which has all arcs labeled with the required flag in the particular direction.

While all examples shown deal with forward conversion functions only, note that transitivity is also given for paths including backward conversion functions as well. In the example of Figure 1, an update of a share object in $sv_3$ will be propagated to $sv_2$ by $cf_{share,2-3} = f_{cf_{share,2-1}} \circ bcf_{share,1-3}$. Due to the automatic composition of conversion functions along a path through the schema derivation DAG, the schema designer has at most to provide $2n-2$ conversion functions for a class with $n$ versions. This is a major improvement in comparison to the approach of Skarra and Zdonik [11, 12]. Details on transitive propagation are given in Lautehm [8].

4.3.2 Holes

There is still one thing which must be mentioned: 'holes are not traversed by the propagation mechanism'. To understand what we mean, consider the history up to $t_3$ shown in the upper part of Figure 8. In this configuration, modifications of Smith in $sv_1$ at $t_5 > t_3$ cannot be propagated to $sv_2$ and transitively to $sv_3$. But imagine, that at time $t_4$ ($t_5 > t_4 > t_3$) object Smith had been deleted from $IAS(sv_1)$ by an application working on top of $sv_3$ (see lower part of Figure 8). This deletion would not be propagated to $sv_2$ because the deletion flag for class Employee in $sv_3$ is not set. Now, Smith would exist in $IAS(sv_1)$ and in $IAS(sv_2)$ but not in $IAS(sv_3)$. This is what we call a hole in the path from $sv_1$ to $sv_3$ regarding the existence of object Smith.

If the described modification of Smith in $sv_1$ occurs at $t_5$, it will not be propagated to $sv_2$ (since the modification flag is set) simply because Smith no longer exists in $IAS(sv_2)$. Therefore, the modification will not be propagated to $sv_3$ either (because $sv_2$'s database has not been updated). This would have been done, however, if Smith had
not been deleted at $t_4$ in $sv_2$. Thus, the status of an object in $IAS(sv_2)$ can depend on the visibility of the same object in $IAS(sv_3)$.

To change the described behaviour, we had two alternatives:

- either we could change the semantics of the modification flag (or offer another flag $m'$), so that Smith would be recreated at time $t_3$ in $sv_2$ and, therefore, the (recreation classified as a) modification would be propagated further

- or we could propagate modifications to all super- and subversions of $sv_1$ which are in the according subgraph, i.e. which have a derivation path where all arcs are labeled with the modification flag in the required direction (e.g. $sv_3$), independent of the existence of the object (Smith) in intermediate schema versions (e.g. $sv_2$).

We have selected the solution where 'holes are not traversed' because it is most straightforward as the propagation is done one step at a time involving one schema version and its direct superversion only. In addition, the two alternatives would result in a complication of both formalization and implementation (see Subsection 4.5). Also, the schema designer would always have to consider indirect superversions as well.

4.4 The Propagation Flags of the Example

We now quickly explain why the propagation flags in our example are set as shown in Figure 1.

As mentioned, a law forces the bank to keep a complete history of share prices. Therefore, $sv_3$ is derived to get $IAS(sv_2)$ as a non-temporary repository. While share objects older than e.g. one month might be removed from $IAS(sv_1)$ as they are no longer required to compute a 30 days average price, such deletions must not be propagated to $sv_2$ (i.e. for $fcfShare,3-1$). There is no backward propagation of shares from $sv_2$ to $sv_1$ (all flags switched off). Obviously, no conversion function is required for such cases.

Shares stored in $sv_1$ and $sv_3$ are information equivalent, as they contain the same semantics. Therefore, it is possible to implement information perceiving conversion functions in both directions ($fcfShare,3-1$ and $bfcfShare,1-3$).

An application that deletes companies is running on top of $sv_1$ and no other application is allowed to delete companies. Therefore, deletions of companies are propagated from $sv_1$ to $sv_3$ but not vice versa. Here, our model also serves authorization requirements between applications on top of different schema versions.

Company objects in $IAS(sv_1)$ do not contain sufficient information for applications on top of $sv_3$. Therefore, the company objects that $sv_3$ gets through the snapshot of $sv_1$ have to be completed manually with founder and location information first. From then on, every object in $sv_3$ company will contain all information required by $sv_3$'s applications. If we had decided to set the modification flag, a change of a company's name in $sv_1$ would trigger a second execution of $fcfCompany,3-1$ (as specified in Subsection 4.1) for this object, thus overwriting the manually included founder and location information with NULL values.

4.5 Implementation Issues

We designed a data structure and algorithms for deferred updates, i.e. not all required conversions have to be performed immediately after each object update or schema derivation. Instead, each object is only converted to a certain schema version $sv$ when it is accessed by an application on top of $sv$.

We call the set of versions of one object visible in the different schema versions the version set of that object. To reduce the physical storage space required for a version set we can make use of deriv-
able object versions because they can be recomputed from other object versions by application of conversion functions.

Assume an object o of a class c is visible in schema versions sv_u and sv_v, sv_v is derived directly from sv_u, and the modification flag is set both for fcfc_u+u and for fcfc_v+v+. If the version of o which has been modified by an application the last time is the one in IAS(sv_u) (sv_u,o), then sv_v,o is derivable from sv_u,o and vice versa. Therefore, in the first (second) case it is sufficient to store sv_u,o (sv_v,o). If sv_v,c and sv_u,c are furthermore information equivalent, i.e. bcfc_u+v, obcfc_u+v(sv_u,o) = sv_u,o and bcfc_v+v, obcfc_v+v(sv_v,o) = sv_v,o, then we can statically decide always to store only the version visible in sv_u for all objects of class c. This decision can be done regarding the storage requirements of sv_v,o in comparison to sv_u,o. But in the more general situation described above we at least have to store the object version which has been modified more recently, i.e. sometimes sv_u,o and sometimes sv_v,o. Obviously, this decision cannot be taken statically.

If the modification flag is set for only one direction, e.g. for fcfc_u+u, we always have to store sv_u,o. Additionally, sv_u,o has to be stored whenever it has been modified after the last modification of sv_v,o.

The strategy minimizing storage requirements is preferable when database requests are mainly updates which have to be propagated to many versions. If storage space is not a problem and read accesses occur more frequently than writes, all versions of one object could also be stored physically, of course. Here, we will present the space minimizing strategy.

Figure 9 shows the storage structure of the situation shown at the left hand side of Figure 10.

![Figure 9: Storage structure of objects.](image)

For a read access we first check, if an object version for sv_v oid is stored physically in the version set of the requested object oid. If this is the case, we simply return this one and are done. Otherwise, we search through the schema derivation DAG to find the object value of a schema version, which can be propagated corresponding to the propagation flags.

Write access: write (svid, oid, value)

For a write access it is not sufficient to simply overwrite the corresponding value because there might exist certain dependencies. Four different cases are shown in the example of Figure 10. As we are now concerned with which object versions are physically stored, we draw a second circle around those schema versions which have the value of the object physically available in the structure shown in Figure 9. For the explanation we assume only forward propagation is switched on.

![Figure 10: Modification of objects.](image)

We say that schema versions that do not have a physical copy of a visible object o are dependent (w.r.t. o) on the closest version, which has a physical copy from which the object value can be propagated. Assuming only forward propagation flags are set, at the left side of Figure 10 schema versions sv_u and sv_v are dependent on sv_v (w.r.t. o). If the object update indicated in Figure 10 takes place, we distinguish four cases:

1) dependent subversion with m-flag (sv_3)
2) dependent subversion with m-flag (sv_3)
3) independent subversion with m-flag (sv_4)
4) independent subversion with m-flag (sv_5)

In case 1, nothing has to be done, but in case 2 the old object value has to be stored in sv_v before the object version of sv_3 is actually overwritten. In case 3, the local copy containing an old value has to be deleted from sv_v, so that future reads in sv_v start searching the correct value in superversion sv_3. As the old value of sv_v (e) might be required by schema versions dependent on sv_v which have the m-flag set, this propagation has to be done recursively starting from the leaves of the schema derivation DAG moving upwards. In case 4, nothing has to be done.

Finally, we simply change the object value of sv_v oid or create a new object version, if it did not exist already (as in the example).

Note that sv_3 and sv_4 became independent while sv_3 and sv_3 now depend on sv_3.

5 Related Research

Bertino [3], Ra and Rundensteiner [10], and Tresch and Scholl [13] propose to simulate schema updates
through views: If all applications are using only views and do not work on the underlying schema directly, a schema update can simply be simulated by changing the views. In this way, the underlying schema never need be updated.

Ferrandina et al. [5] use conversion functions to propagate schema updates to the instance level but allow only one schema version at a time. This results in some problems: Firstly, every application has to be adapted to every schema update immediately in order to work continually. Secondly, interdependencies between schema updates in different classes cause serious problems with the physical deletion of instances which conformed to the previous state of the schema. Thirdly, the proposed algorithms do not guarantee to preserve time-equivalence, as the result of a deferred update can differ from an immediate one.

Kim and Chou [6] propose a schema versioning concept for the Orion prototype OODBMS. Transient and working schema versions are organized in a version derivation hierarchy. Each schema version has an associated set of objects called its access scope which consists of a direct and an inherited part. A set of seven rules governs the system also regarding inheritance of instances from their creator schema version to its subversions (called descendants) and the access to and deletion of instances of the access scope of the corresponding schema versions.

However, the proposal lacks some flexibility: There is only the choice to inherit either the complete database from the direct supervision (called parent) or nothing. In addition, attribute values of inherited objects can be changed, but the change will always be overwritten by subsequent changes of the same object in a supervision. Finally, only forward propagation of instances (i.e. from old to new schema versions) is supported.

Skarra and Zdonik [11, 12] propose versions of types as a flexible way to handle schema evolution. When an application expecting a new version of a type is accessing an object which is stored in the database according to an old version of the type (or vice versa), problems can occur in two ways: If the constraint of the type has been strengthened, a read access might return result values outside of the new type version (reader’s problem). On the other hand, if the constraint has been relaxed, a write access might fail, because the new value is outside the domain of the stored object’s type version (writer’s problem). The authors propose a version set interface which includes all properties defined by any version of the type. Reader’s and writer’s problems are solved by exception handlers which are used if the object access results in an undefined (an attribute is not defined for a given type version) or an unknown (a value is outside an expected domain) error. These error values can be replaced by useful results which are computed by user defined pre- and post-handlers before the application gets them.

The approach is similar to the views concept given that all values of an object visible in different type versions can be computed from each other using the exception handlers. Furthermore, there is no possibility to restrict the visibility of an object to a certain subset of the versions of a type or to limit the propagation of object values to newer or older versions. Another disadvantage is that schema evolution cannot be done locally, because no automatic composition of handlers is done to compute transitive conversions. Therefore, the introduction of a new type version requires the addition of handlers for all existing type versions, not only for one.

Monk and Summerville [9] present an approach which supports schema evolution by versioning at the class level. A class can have different versions which are derived from each other in a linear sequence, along which instances can be converted from old to new schema versions (using update functions) and vice versa (using backdate functions). In this way, also applications developed for old schema versions can access instances which are created and modified by applications working with newer class versions.

While the approach allows to use the same attribute name with different semantics in different class versions (e.g. to change the unit of measurement from inch to cm), the values of the corresponding instances cannot be completely independent of each other. Every time an attribute value is changed the update and backdate functions are used immediately to update the instance in all other class versions.

6 Conclusion and Future Work

We have presented a model of schema evolution which allows applications to run on different versions of one schema. The portions of the complete database which are visible in the different schema versions can share data by using a propagation mechanism which can be adapted flexibly according to the schema designers needs. In this way, our approach enables the evolution of a schema without the requirement to adjust all running applications immediately which would prevent most schema improvements in practice. Instead, applications can be adapted to new schema versions whenever this seems to be advantageous.

One of our design goals has been to allow local schema evolution, so that a schema designer does not have to know the complete schema derivation DAG to derive a new schema version and to specify the required conversion functions and propagation
flags. Instead, it is sufficient to only consider the direct superversion and to specify the relationship between the new schema version and its direct superversion.

Currently, we are working on the COAST prototype implementation consisting of three main modules: an Object Manager and a Schema Manager that work as in most systems except that ours can handle versioned objects and schemas, and a Propagation Manager that completely hides the deferred implementation style and allows extra conversion functions (see Lautemann [8]) to increase the amount of object semantics that can be propagated between different versions of the schema derivation DAG.

6.1 Acknowledgements
We would like to thank Ling Liu for the useful discussions during which most of the fundamental ideas presented in this paper were brought up.

References


