First publ. in: Proc. of the 1st ECOOP Workshop on Object-Oriented Databases, Lisboa, Portugal, June 1999

Outdating Outdated Objects

Holger Riedel *

Abstract

In many application scenarios the database is changing quite rapidly. Because the management of such data is rather expensive and cumbersome, many applications like data warehouses only store some aggregations of it. We describe a strategy how the evolution of object databases can be controlled by enhanced versioning. Therefore, we extend the notions of ODMG in order to capture temporal versions. We analyze the consequences for the consistency notions and the semantics of queries. Moreover, we sketch a framework how these concepts can be implemented.

1 Introduction

In the age of terabyte databases, a lot of information is collected by many applications. Although *storing* such masses of data is no problem in nowadays database management systems, the *processing* of interesting queries poses many restrictions. Especially when temporal data is recorded in the database, the following phenomen can be observed:

As information gets old:

1. its worth and usefulness shrinks quite rapidly;
2. the costs of its maintenance increase for updating and querying current objects and for changing the database schema or the physical representation.

Working with such outdated data is insufficiently supported in current research and products. A rather simple solution is proposed for temporal databases using the concept of *vacuuming* which was invented to describe the process of destroying a certain amount of historical data in a temporal database. Up to now, only ideas are presented how to design useful update operators for vacuuming relational databases [Sno95], but no support for automatic control of such issues or efficient implementation concepts are given. Also the possibilities and restrictions of object-oriented approaches are not discussed in this context.

In *data warehouses* [Bar97] the problem of historical data is solved in a much simpler way. In order to decrease the amount of data to be handled later on, only data which seems useful in the future is taken from the operational database into the data warehouse database. Therefore, the decision of the usefulness is made only on structural elements, not on the age of the specific data. Vacuuming the data warehouse is left to the application administrator without further support by the database management system.

Another area with similar problems is *versioning* as present in several OODBS [KM94]. Version models were originally invented for design transactions present in engineering applications to describe complex changes like the construction of an engine on the database level. Versioning models support object-oriented concepts including complex types. Also the migration of objects within versions and the different possibilities how versions can be connected is well-understood. On the other hand, no work has been done to put these ideas at work for temporal data and

*address: Fakultät für Mathematik und Informatik, Universität Konstanz, D-78457 Konstanz, Germany, email: Holger.Riedel@uni-konstanz.de

especially for the problems sketched in this paper. Also the current ODMG standard [Cat97] does not handle versioning at all.

In order to solve the problem described above, we propose a temporal versioning schema for object-oriented databases. Therefore, arbitrary parts can be marked for deletion according to a temporal condition. These parts are put into different versions organized in a linear order. Thus, each object of such a versioned class walks through these versions in a fixed way. Although this leads to versioning on the object-level, arbitrary OQL queries including additional temporal constructs can be used, either ignoring or explicitly using the versions of a specific class. We clarify the semantics of such queries in this context. Then the problem how partially outdated objects should be present in the semantics of queries has to be solved. Although null values cannot be avoided in this place, we show how to handle the problem in a promising way. Implementing such an approach is a rather difficult task. We propose a flexible architecture supporting lazy and eager evaluation schemes for version migration, flexible object-oriented storage structures, and enhanced query transformers and optimizers.

Outline of the Paper. In the next section we present the extensions to the ODMG object model describing temporal-versioned objects, while the semantics of queries is discussed in section 3. Several alternatives to realize these concepts are sketched in section 4.

2 Temporal versioning

2.1 The object model

In our approach, we extend an arbitrary ODMG database schema by information on versions for some classes. The ODMG data model [Cat97] describes objects along with their complex types. Arbitrarily complex objects can be modelled using several type constructors (like `set`, `struct`, `list`, `bag`, `array`). Also the distinction between objects (with an value-independent identity) and literals (without oid) is supported. The schema is described by an ODL notation.

In [RS96, RS97] we gave a formal definition of the ODMG model. So we can formally argue on inheritance and queries in this approach. Due to space limitations, we do not repeat the formalism here, but describe informally the important aspects w.r.t. objects and classes. The set of all (abstract) objects is denoted by $\mathcal{D}_{Object}$. The *extent* of a class C is a set of objects, denoted by $\sigma(C)$. The type *type(C)* of a class C is described as a set of global functions using the notion $[f_1, \ldots, f_n]$. These *object functions* can recursively be structured using structs, sets, bags, lists, and arrays. There is a subtype hierarchy (denoted by $\tau_1 \preceq \tau_2$) between such types, describing that the type τ_1 is a subtype of τ_2. The subtype hierarchy is a partial order constructed by the following rules:

- collection types are in a subtype hierarchy if its element types are in a subtype hierarchy. Formally:

$$\text{[SETS]}\ \frac{\tau_1 \preceq \tau_2}{\{\tau_1\} \preceq \{\tau_2\}} \qquad \text{[BAGS]}\ \frac{\tau_1 \preceq \tau_2}{\{\!\{\tau_1\}\!\} \preceq \{\!\{\tau_2\}\!\}}$$

$$\text{[ARRAYS]}\ \frac{\tau_1 \preceq \tau_2}{\tau_1[\cdot] \preceq \tau_2[\cdot]} \qquad \text{[LISTS]}\ \frac{\tau_1 \preceq \tau_2}{\langle\tau_1\rangle \preceq \langle\tau_2\rangle}$$

- a struct is a subtype of another struct, if at least it has the same components and the types of these common components are in a subtype relationship:

$$\text{[STRUCTS]}\ \frac{\tau_1 \preceq \tau_1', \ldots, \tau_n \preceq \tau_n'}{(L_1 : \tau_1, \ldots, L_n : \tau_n, \ldots, L_m : \tau_m') \preceq (L_1 : \tau_1', \ldots, L_n : \tau_n')}$$

- a object function is a subtype of another object function if it obeys the contra-variance principle:

```
CLASS Product AVAILABLE UNTIL DELETED {
  VERSION P1 AVAILABLE 1 YEAR {
    name      string;
    desc      string;
    details   set<struct<
                   detail:string,
                   added_by:Engineer>>;}
  VERSION P2 AVAILABLE ON DEMAND {
    name      string;
    details   set<struct<
                   detail:string>>;}
  VERSION P3 AVAILABLE UNTIL DELETED {
    name      string;}};
```

```
CLASS Workpiece AVAILABLE UNTIL DELETED {
  VERSION W1 AVAILABLE 1 YEAR {
     serial_no   number;
     is_of       Product;
     made_by     Workman;}
  VERSION W2 AVAILABLE UNTIL DELETED {
     serial_no   number;
     is_of:      Product;}};
```

Figure 1: A temporal-versioned database schema

$$[\text{FUNS}]\ \frac{\tau_1^{dom} \preceq \tau_2^{dom},\ \tau_2^{rng} \preceq \tau_1^{rng}}{\tau_2^{dom} \to \tau_2^{rng} \preceq \tau_1^{dom} \to \tau_1^{rng}}$$

2.2 Versioned database schemas

In order to model temporal versioning in an ODL schema, we add the following:

- A temporal condition can be added for each part of the database schema describing how long this data is available. Therefore, the following constructs can be used:
  1. a description of a fixed temporal interval using the notions of SQL2 [MS93], e.g. `2 YEAR`;
  2. `ON DEMAND`: this part will be deleted from this version on a dedicated request according to the necessities of the application;
  3. `UNTIL DELETED`: this part will be stored until the corresponding object is deleted in the database.
- Using the notions above, each class can be versioned using the notion

  class_name `AVAILABLE` <*time spec*> {
  `VERSION` *version_name_1* `AVAILABLE` <*time-spec*> {*attribute-list*}
  ...
  `VERSION` *version_name_n* `AVAILABLE` <*time-spec*> {*attribute-list*}.

Example 1 Our small application describes the production of workpieces for certain products. A part of the conceptual schema is given in figure 1. We differ slightly from ODMG-ODL by describing complex structured types in a more intuitive way. Workpieces are temporally versioned into two versions `W1` and `W2`. Each workpiece recorded in the database is stored in `W1` for the first year, and afterwards in `W2` until deleted by the application. The global information of a workpiece is captured by the class `Product` which is versioned using three stages. An interesting aspect is the versioning of the class `Product` for the complex attribute `details`, because the information of the responsible `Engineer` is no longer present in `P2`. □

2.3 Consistency

2.3.1 Additional query operators

We introduce two query operators which transform objects and literals according to the type hierarchy:

- A *cast* operation is an extension of a relational projection, transforming an object or literal into an instance of an arbitrary supertype.
- The *null-extend* operator does the opposite. An object or literal is transformed into an instance of a subtype setting additional parts to a specific null value. This operator is not present in ODMG-OQL, but added for the purposes of our versioning approach.

Cast. The cast operator is an OQL query operator extending the relational projection operator recursively to arbitrarily structured types. It can be applied to collection-valued instances as well as to single objects. The cast operator has two operands, an extent E of an arbitrary ODL-type τ_1 and a type expression τ_2 denoting an arbitrary supertype of τ_1 according to the definitions given above. Then the cast operation `(`τ_2`)(E)` casts each object of E of type τ_1 to the type τ_2. Therefore, components not present in τ_1 are eliminated by a recursive reduction given below. Collection-valued components are treated as follows:

- there are no duplicates in sets;
- arrays and lists are reduced for each element;
- The cardinality of a tuple (denoted by the function *card*) in the resulting bag is calculated by the sum of the cardinalities of the tuples which are used for its computation.

The new instance of E ist given by $[\![(\tau_2)\ E]\!] =$

type of τ_2	*result*		*further conditions*
basic type	v	:	$[\![E]\!] = v$
struct	$(L_1 : (\tau_{2,1})\ c_1, \ldots, L_m : (\tau_{2,m})\ c_m)$	:	$[\![E]\!] = (L_1 : c_1, \ldots, L_m : c_m, \ldots, L_n : c_n) \wedge \tau_2 = (L_1 : \tau_{2,1}, \ldots, L_m : \tau_{2,m})$
set	$\{t' \mid \exists t \in [\![E]\!] \wedge (\tau_2')\ t = t'\}$	:	$\tau_2 = \{\tau_2'\}$
bag	$card(t', [\![(\tau_2)\ E]\!]) = \sum_{(\tau_2')t=t'} card(t, [\![E]\!])$	:	$\tau_2 = \{\!\{\tau_2'\}\!\} \wedge t' \in [\![\ \{\!\{\tau_2'\}\!\}\ ]\!]$
list	$\langle(\tau_2')\ c_0, \ldots, (\tau_2')\ c_n\rangle$	:	$[\![E]\!] = \langle c_0, \ldots, c_n\rangle \wedge \tau_2 = \langle\tau_2'\rangle$
array	$[(\tau_2')\ c_0, \ldots, (\tau_2')\ c_n]$	:	$[\![E]\!] = [c_0, \ldots, c_n] \wedge \tau_2 = [\tau_2']$
function	$o \rightarrow (\tau_2')\ v$	:	$[\![E]\!] = o \rightarrow v \wedge \tau_2 = \mathcal{D}_{Object} \rightarrow_{fin} \tau_2'$
object type	$\sigma([f_1, \ldots, f_m])$	:	$[\![E]\!] = \sigma([f_1, \ldots, f_m, \ldots, f_n]) \wedge \tau_2 = [f_1, \ldots, f_m]$

The *null-extend* operator. The operator operator `ne` (as a shorthand for `null-extend`) can be applied to each instance E of type τ_2 which must be supertype of τ_1. Then, `ne[`τ_1`](`E`)` results in an instance of type τ_1, where missing components of object functions or tuple components are set to a specific null value. If complex types are used, the operator will be applied to its components recursively. Because extending a struct does not generate new duplicates within sets and does not change the cardinality of a struct within a bag, all four collection-valued type constructors are treated the same way: the `ne` operator applied to one element of the collection exactly generates one distinct result element.

The new instance is built according to $[\![\ \texttt{ne}[\tau_1](E)\ ]\!] =$

type of τ_1	*result*		*further conditions*
basic type	v	:	$[\![E]\!] = v$
struct	$(L_1 : (\mathtt{ne}[\tau_{1,1}](c_1), \ldots, L_m : \mathtt{ne}[(\tau_{1,\mathtt{m}}),$	:	$[\![E]\!] = (L_1 : c_1, \ldots, L_m : c_m),$
	$L_{m+1} : \mathtt{NULL}, \ldots, L_n : \mathtt{NULL})$		$\wedge\ \tau_1 = (L_1 : \tau_{1,1}, \ldots, L_n : \tau_{1,n})$
set	$\{t' \mid \exists t \in [\![E]\!] \wedge t' = [\![\mathtt{ne}[\tau_1'](E)]\!]\}$	:	$\tau_1 = \{\tau_1'\}$
bag	$card(t', [\![(\tau_1)\ E]\!]) = card(t, [\![E]\!])$	:	$\tau_1 = \{\!\{\tau_1'\}\!\} \wedge t' \in [\![\ \{\!\{\tau_1'\}\!\}\ ]\!] \wedge$
			$\tau_2 = \{\!\{\tau_2'\}\!\} \wedge\ \ t = (\tau_2)(t')$
list	$\langle \mathtt{ne}[\tau_1'](c_0), \ldots, \mathtt{ne}[\tau_1'](c_n) \rangle$	:	$[\![E]\!] = \langle c_0, \ldots, c_n \rangle \wedge \tau_1 = \langle \tau_1' \rangle$
array	$[\mathtt{ne}[\tau_1'](c_0), \ldots, \mathtt{ne}[\tau_1'](c_n)]$	:	$[\![E]\!] = [c_0, \ldots, c_n] \wedge \tau_1 = [\tau_1']$
function	$o \rightarrow \mathtt{ne}[\tau_1'](v)$	:	$[\![E]\!] = o \rightarrow v \wedge$
			$\tau_1 = \mathcal{D}_{Object} \rightarrow_{fin} \tau_1'$
object type	$\sigma([f_1 : \mathtt{ne}[\tau_{1,1}](c_1), \ldots, f_m : \mathtt{ne}[\tau_{1,m}$	:	$[\![E]\!] = \sigma([f_1 : c_1, \ldots, f_m : c_m])$
	$](c_m), f_{m+1} : \mathtt{NULL}, \ldots, f_n : \mathtt{NULL}])$		$\wedge \tau_1 = [f_1 : \tau_{1,1}, \ldots, f_n : \tau_{1,n}]$

2.3.2 Consistency of the database

Consistency of a versioned database schema. Well-defined versioned database schemas have to obey the following consistency constraints:

1. All constraints of the object schema which do not refer to versioning must be obeyed as usual, i.e., attributes reference to other classes, not to specific versions of these classes.
2. The type of a class C is the type of its first version.
3. The versions of a class are ordered in a linear order[1] and each version V_j describes less information than its predecessor V_{j-1}, i.e., the type of V_j is a supertype of V_{j-1}.
4. According to the goal of the paper, attributes present in different versions model the same information. Thus, all attributes can be seen as attached to the class and are restricted for specific versions.
5. The `AVAILABLE` specification of the class must be covered by the `AVAILABLE` specifications of its versions, i.e., each object of a class is stored in exactly one version for each point in time.
6. If a version contains an attribute referencing to another class, the availability of the version expressed by the `AVAILABLE` clause must be a subset of the availability of the referenced class.

Consistency of the extent of a versioned database. In our approach, an object migrates through different versions, but its attributes are global. Thus, on a conceptual level, the different versions can be seen as views obeying temporal conditions on the non-versioned database schema. Therefore, updates are always performed on the class level, and propagated to the appropiate version, if the corresponding attribute is present there. We stress on the goal of our approach that the different versions are not utilized as queryable views, but as a means to clean up the database according to the specification in the database schema. As described in section 4, this point of view allows the realization of an internal layer of the database supporting versions and the migration in a flexible manner.

Extent of a class. Later on, queries can be formulated accessing a class insteadof a specific version. This can be used to obtain transparency on the user level. Therefore, it is necessary that the extent of a class can be derived as a query using its versions. This can be obtained by a union of the extents of the different versions using the `ne` operator. Thus, in this case, the combination of `ne` and `union` works like an outerunion on complex structured objects.

[1] Other orderings supporting branching and/or merging of versions are possible, but need further specification for correct database schemas.

The extent of a class $\sigma(C)$ with versions $V_1, \ldots, V_n$ and $type(C) = type(V_1) = \tau_1, type(V_2) = \tau_2, \ldots, type(V_n) = \tau_n$ can be built by the query

$$\sigma(C) = V_1 \texttt{ union ne}[\tau_1](V_2) \texttt{ union } \ldots \texttt{ union ne}[\tau_1](V_n).$$

Example 2 The database schema of figure 1 is consistent to the notions above, because the version types of both `Product` and `Workpiece` are subtype hierarchies and the referenced class `Product` uses the option `AVAILABLE UNTIL DELETED` which covers the availability conditions of the versions of `Workpiece`. The attributes `added_by` and `made_by` refer to non-versioned classes and the usual referential integrity will be enforced by the ODBMS in these cases. □

Migration. Also the walk of an object through the different versions can be described using the cast operator. Formally:

> Let o be an object of a version V_j with type τ_{V_j} of a class C. When migrating to version V_{j+1} (with type τ_{j+1}), the extent of o is changed according to
>
> $$\sigma_{V_{j+1}}(o) := (\tau_{V_{j+1}})\ (\sigma_{V_j}(o)).$$

3 Semantics of queries

Given the temporal versioning schema, queries can be formulated using the features of ODMG-OQL. The syntactical formulation of queries can explicitly access specific versions or only use transparent class accesses which can be translated to version accesses using the `ne` operator as described in the last section.

Clarifying the semantics of queries, we have to consider that a query always asks for information as present in the conceptual schema. Thus, issueing a query for a versioned class leads to some semantical problems, because the database management system makes some parts of the data invisible to the user (and might delete these parts as well).

Example 3 We are asking for the available `details` for existing workpieces using the query

```
select w.is_of->details from Workpiece w.
```

Using the OQL type checker using the formalization from [RS96, RS97], the result type is

```
bag<set<struct<detail:string,added_by:Engineer>>>.
```

But according to the temporal versioning schema of the classes `Workpiece` and `Product` and the current available objects refering to products of `P2` have only restricted information on `details`, while workpieces of `W3` have no information on `details` at all.

Defining a semantics for queries, this problem can be handled in different ways:

1. Such information will be handled as non-existent, thus any tuple or object with such attributes are completely removed from the result. This is a means to avoid null values in the result in a very stringent way.
2. Such information will be denotated by SQL's standard null-value `NULL`.
3. Such information will be denotated by an additional null-value `HISTORICALLY_DELETED`, denoting that there was information present which no longer exists.

While simple standard queries can go along with all of the three approaches quite easily, the picture gets rather messy when aggregations are involved. We look at the query

```
select count(w.is_of->details) from Workpiece w
```

What is the correct answer for workpieces of `W2` referencing to products of `P3`?

As we see, alternative (1.) is not usuable at all in a data model supporting versioning on attributes or parts of attributes, because then objects with partial information available are dropped completely from the result before the aggregates are evaluated. □

Thus, at least SQL's approach of using `NULL` is necessary to capture such cases. On the other hand, it seems necessary to distinguish between arbitrary `NULL`s in the database and the temporal versioning in many applications.

Semantics of a query. Let q be a query accessing the classes $C_1, \ldots, C_n$ with types $\tau_1, \ldots, \tau_n$. Each C_j has the versions $V_{j,1}, \ldots, V_{j,i_j}$. Then

$$[\![q(C_1, \ldots, C_n)]\!] := [\![q(V_{1,1} \texttt{ union ne}[\tau_2](V_{1,2}) \texttt{ union } \ldots \texttt{ union ne}[\tau_1](V_{1,i_1}), \ldots, V_{n,1} \texttt{ union ne}[\tau_n](V_{n,2}) \texttt{ union } \ldots \texttt{ union ne}[\tau_n](V_{n,i_n}))]\!]$$

Remark. The substitution of C by its versions in arbitrary OQL queries is well-defined because OQL supports orthogonality for any kind of queries.

Accessing versions. As an add-on to the querying capabilities of ODMG-OQL, direct access to a version is possible using the notion *class.<version_name>*. Because each version has its own type description, the type of a version access is the type of the version.

Example 4 Accessing only the details for workpieces of version `W2` can be done by

```
select v.is_of->details from Workpiece.W2 v.
```

□

4 Implementation

4.1 Overview

Designing an efficient implementation strategy, at least the following topics have to be considered:

- How do we store the different versions of a class?
- How do the objects migrate?
  - Do they move on the physical level (and when)?
  - Are specialized indices maintained?
- How must query processing be changed for such an environment?

We propose a flexible architecture sketched in figure 2 to solve these problems. Therefore we combine concepts of lazy and eager migration of versions, flexible physical storage structures of object-oriented databases, and elaborated techniques of query processing in standard databases in a specific way.

This architecture has the following major parts:

- Propagation of versioning: The database schema describes which conditions have to be fulfilled for the migration of an object. On the other hand, we propose a multi-leveled system with indepedent layers. Then the versioning on the physical level can be
  - *eager*: the system obeys when objects are affected by the constraints of the database schema and enforces the transition into the next version;

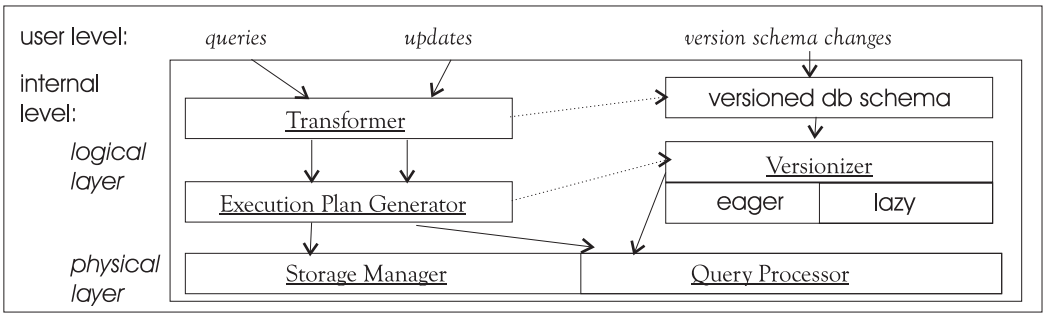

Figure 2: Architecture

  - *lazy*: transitions are delayed until enforced by user requests or the objects are needed for queries or updates. We also describe a *deferred query processing* strategy evaluating queries on outdated versions and correcting the results according to the temporal information available in the database schema.

- The objects can flexibly be stored on the physical layer. Capturing temporal versioning, we propose especially the following alternatives:

  1. *no_split*: Objects are stored once in the class extent. The current version is marked by an additional attribute `version_`<*version_name*> which is set by the `Versionizer`.
  2. *indexed_no_split*: As 1., but additionally an adequate temporal index [Ber97] referencing for each object to its current version and a timestamp of the last migration is present.
  3. *partition by version*: There are different storage areas for each version and objects migrate through these according to the versioning conditions in the database schema.

- Updates and queries are evaluated using the the concepts above. Efficiency and optimization potential of the generated execution plans depend on the needs of the applications and the aspects discussed above.

4.2 Versioning

In the following paragraphs we describe the possibilities of the approach by some examples.

Eager Versioning The changes described in the database schema are evaluated as soon as possible, for instance by the use of fixed polling intervals depending on the specifications given in the database schema. Further optimizations are possible by the use of temporal indices indicating which objects have to be migrated next. Another optimization can be obtained by restricting the possibilities of putting updates into the database only in fixed intervals (e.g., it might be useful to record information on employess only once a month). The major advantage of eager versioning is its applicability to all the types of `AVAILABLE` specifications present in the conceptual database schema. Also queries can be evaluated without further checks in this framework. On the other hand, bad performance can be expected in circumstances when short intervals are specified, many updates are present, or queries are used infrequently.

Example 5 We store `Product` and `Workpiece` using the *no_split* alternative. Then the query

```
select w.serial_no, w.is_of->details from Workpiece.W2 w
```

can be evaluated using the following steps:

1. The `Query-Transformer` uses the `AVAILABLE` conditions of `W2` to access the attribute `details` only for products of `P2` and `P3`, because according to the given versioning schema, the products for workpieces of `W2` had also to be migrated from `P1`.

2. The `Execution-Plan-Generator` uses the information of *no_split* to check the versions by using the attributes `version_W2`, `version_P2`, and `version_P3`.

3. Because eager versioning is used, the `Execution-Plan-Generator` does not access the `Versionizer`.

4. Because `W1` and `P1` use a fixed interval in the `AVAILABLE` clause, further optimizations are possible, replacing the access to the `version_`v_j attributes by simple temporal conditions. The usefulness of this step depends on the underlying cost model.

□

Lazy Versioning. The `Versionizer` does not check the versions via polling, but only when queries or updates use this part of the database schema according to the information generated by the type checker. Therefore, additional mechanisms have to be added to the execution unit to enforce the necessary action when queries or updates are launched. When using the `ON DEMAND` option, some operations have to be done anyway in the case of a versioning request, which might be a handicap for this approach in such circumstances. On the other hand, updating of the physical representation can significantly be reduced in comparison with eager versioning in scenarios where short intervals for versioning are specified.

Example 6 We refer to the last example, but now we use lazy versioning for `Product`. The correct query result can be obtained in the same way as before, but a prior call of the `Versionizer` is necessary to set the `version_P`$_j$ attributes correctly. Using a temporal index, this additional effort can be reduced to the absolute necessary. □

4.3 Advanced query processing

An interesting advantage of the use of the temporal specification in the database schema can be obtained when it is combined with the query processor to simplify the accesses to the different versions. Especially, when temporal query languages like TSQL2 [Sno95] are used, often temporal conditions will be part of the query, which can be used to simplify the access plan by the use of a deductive component establishing subsumptions on the conditions of the queries and the temporal specifications of the database schema extending the work done in relational query languages [Ull89]. We do not work out the details here, but focus on another possibility how the temporal specifications can used to avoid migration of versions at all. We call this approach *deferred query processing*, because the migration of versions will be avoided as long as possible and additional query parts are introduced to give a correct query result.

Deferred query processing. The basic idea can be described as follows:

1. Each version of an object describes less information than its predecessor.

2. Thus, a query which needs a certain version v_j of an object can obtain it from any earlier version if it can be deduced that the object belongs to v_j when accessed. This can be achieved by the use of the temporal conditions in the database schema or the settings of the `version_`v_j attribute.

3. The complete evaluation schema for a given query looks as follows:

   (a) Evaluate the query without prior triggering the `Versionizer`.

   (b) Extend the intermediate result types by the versioning attributes `version_`<*version_name*>.

(c) If possible, generate the final result by deleting any information belonging to objects which do not fulfill the requirement for the query according to their versioning attribute.

(d) At some stages of the query evaluation, the check whether an object belongs to a certain version cannot be delayed (e.g. for computing aggregates). If such a node is reached in the evaluation schema, trigger the `Versionizer` for the objects processed for this node.

This evaluation schema will be useful, if it is rather expensive to check the `Versionizer` and there are several restrictive selections at prior stages of the evaluation of the particular query. Thus, designing an appropiate cost model is a crucial demand for the whole approach.

Example 7 We assume partition-by-version storage combined with lazy versioning for both classes `Product` and `Workpiece`. Then the query

```
select w.name, p.details
from Workpiece w, Product.P3 p
where (CURRENT_DATE - w.version_W1) > 2 YEAR
        and w.is_of=p and p.name='Frigo XP100'
```

retrieves all workpieces which have been stored in `Workpiece` for at least 2 years and the referenced product "Frigo XP100" is stored in `P3`. Now the query processing can act in the following way:

- It can be derived via temporal subsumption that the requested workpieces have to belong to version `W2`.
- Because lazy versioning is used for the class `Workpiece`, objects of the version `W1` have to be considered using the condition `(CURRENT_DATE - w.version_W1) > 2 YEAR` as an additional restriction.
- Because products are migrated to `P3` by the `ON DEMAND` clause in the specification of `P2`, the attribute `version_P3` is always correctly set even in the case of lazy versioning. Thus no further checks on `P1` or `P2` are necessary.
- The selection condition `p.name='Frigo XP100'` can be used as a further restriction for accessing `P3`.

□

We close our description of the implementation aspects by focusing on the advantages gained for the design and maintenance of the internal representation of the database. In our approach, the internal layer can be changed without any affects to the semantics of the database and queries:

- Changes of the physical representation (e.g., adding a temporal index) can be integrated on the fly.
- Old and new versions of the same class can be supported in different ways (e.g. using additional access structures only for specific versions).
- Changing the versioning policy on the conceptual level can easily be integrated into an existing database allowing further versions being specified in the conceptual schema. Even changing the temporal conditions of specific versions will easily be handled if the conditions are sharpened (lowering the temporal conditions has to be forbidden, because then the DBMS might need to access objects which have been deleted earlier).

5 Conclusion

We presented a framework how information on objects or on parts of objects can be marked in the database schema as outdated. Because several temporal conditions can be used in parallel, a multi-level staging can be described how the stored data can be reduced using several versions of a class. We discussed the semantics of such a database schema and its impact on the use of the database. We demonstrated how the semantics of queries undergoes some changes in order to keep the global picture for the user or application. We sketched several implementation frameworks for this approach showing how the temporal specification of the database schema can be used to simplify query processing in a flexible way supporting eager and lazy strategies.

The presented framework leaves an open field for future work:

- It seems reasonable to add a broader class of temporal conditions into the availability specification. Even non-temporal conditions might be interesting, although then the demand that versions are walked through in a linear order might be difficult to handle.
- In order to simplify the migration step, we used the constraint that versions are linearly ordered. Obviously more complex scenarios of the version graph, like a tree-shape or an acyclic graph are quite useful. Therefore additional details have to be fixed in the specifications of such versions describing the new version where an object should be placed afterwards. If an acyclic version graph is used, then also the merge step will need a closer look, because it must be ensured that the different branches to not lead to inconsistent availability specifications.
- Adequate cost models for the different implementation alternatives are necessary to do the right decisions on the implementational level.
- Also the possibilities to deduce subsumptions of temporal conditions in the query language and the version specification seem worthwile for further analysis in order to optimize the query processing strategy.

References

[Bar97] R. Barquin. *Building, using and managing the data warehouse.* Prentice Hall, 1997.

[Ber97] E. Bertino. *Indexing techniques for advanced database systems.* Kluwer Academic Publishers, 1997.

[Cat97] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-2.0.* Morgan Kaufmann, 1997.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice-Hall, 1994.

[MS93] J. Melton and A. Simon. *Understanding the new SQL: A Complete Guide.* Morgan Kaufmann, 1993.

[RS96] H. Riedel and M. H. Scholl. *The CROQUE-Model: Formalization of the Data Model and Query Language.* Technical Report 23/96. Dept. of Mathematics and Computer Science, University of Konstanz, 1996.

[RS97] H. Riedel and M. H. Scholl. A Formalization of ODMG Queries. In *7th Intl. Conference on Data Semantics (DS-7)*, 1997.

[Sno95] R. Snodgrass, editor. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.