Version Management for Scientific Databases

I-Min A. Chen[1], Victor M. Markowitz[1],
Stanley Letovsky[2], Peter Li[2], and Kenneth H. Fasman[2]

[1] Information and Computing Sciences Division
Lawrence Berkeley National Laboratory, Berkeley, CA 94720
[2] Division of Biomedical Information Sciences
Johns Hopkins University, Baltimore, MD 21205

Abstract. Scientific databases are used to accession objects representing the results of scientific inquiry, such as genes and DNA sequences. These objects must have stable identifiers that can be used as references in scientific papers and other databases. The requirement for stable object identifiers, however, conflicts with the tendency of scientific data to evolve over time. We present in this paper version management facilities that allow scientific databases to achieve a balance between stable object identifiers and evolving data.

1 Introduction

Scientific databases are increasingly being used to accession objects representing the results of scientific inquiry, such as genes and DNA sequences. Accessioning involves providing these objects with stable identifiers that can be included as references in publications (e.g., journal papers) and other scientific databases. This requirement for stable object identifiers can conflict with the tendency of scientific data to evolve over time. By the time a scientist tracks down the scientific object recorded in the database, this object may have disappeared, or changed to the point where it no longer supports the claims made in the associated publications. The traditional literature does not have this problem; a paper once published remains unchanged, even if its claims have been refuted by later papers. A printed paper, however, cannot indicate to scientists whether the claims in the paper were subsequently refuted. Conversely, a database can maintain data representing the current view of scientific observations; however maintaining the stability of references to database objects as the underlying scientific data evolve is a problem.

In this paper, we present version management facilities that allow scientific databases to achieve a balance between stable object identifiers and evolving data, by providing support for:

1. maintaining stable object identifiers, where an identifier represents a particular state in the history of that object but is not rendered meaningless by changes to the object;
2. reconstructing the change history of objects in order to support queries that involve past as well as current object versions;

3. merging objects when they are discovered to represent the same real-world entity;
4. splitting an object that is found to represent more than one real-world entity.

Although these facilities are examined in the context of scientific databases, they can be applied in non-scientific database environments as well.

Versions have been examined in numerous papers dealing with general version management (e.g., [4, 7, 18, 19, 22]), or with versioning for special applications such as office information systems [2], engineering databases [9], CAD applications [1, 6, 13, 15, 16], and multimedia databases [20, 21]. However, the version management facilities mentioned above are not addressed by the versioning mechanisms described in the literature.

The first problem considered in this paper regards keeping track of current (*default*) object versions. Default versions are convenient for supporting generic (rather than version specific) references [4, 16, 19]. For example, suppose that a clinical syndrome, *breast cancer*, is associated with gene *brca1*. If the syndrome references a specific version of *brca1*, then each time *brca1* is revised as new information about the gene is discovered, a new version of the *breast cancer* syndrome needs to be created in order to reflect the change in the referenced gene. If other objects refer to *brca1*, then they may also need to be revised. The concept of default versions allows modeling this situation in a more efficient way using a *generic reference* from syndromes to the current versions of genes. In order to reconstruct a past database state at time t, the default versions of all versioned objects must be recorded so that past generic references to a versioned object x can refer to the default version of x at time t instead of referencing the last specified default version of x. We show in this paper that reconstructing the change history of objects in the context of generic references requires maintaining a history of default versions in addition to, and separate from, the history of each version.

The second problem addressed in this paper involves managing versions in class hierarchies. Version management for class hierarchies has to take into account the fact that an object can belong to multiple classes and therefore its versions can involve values of both local and inherited attributes. For example, consider data on authors of scientific references and citations. Let class `AUTHOR` be a subclass of class `PERSON`, and suppose that at time t_1 `PERSON` contains a version of object p; at time t_2, p becomes an object of class `AUTHOR`, and a new version of p is created. The version of p before t involves only the values of the attributes of `PERSON`, while the version of p after t involves the values of the attributes of both `PERSON` and `AUTHOR`. We show that if an object x belongs to multiple classes of a class hierarchy, then determining the values of all the versioned attributes of x at any given time t, requires a mechanism for combining versions of x across the classes containing x.

Finally, we consider the requirement for version derivations spanning different objects. The versioning mechanisms discussed in the literature do not support merging versions of different objects into a version of another object, or splitting a version of an object into versions of new objects. Such object merging and splitting is required for modeling the evolution of database objects representing imperfectly known real-world entities, such as those resulting from scientific observations.

Consider, for example, a class `GENE` representing genes and having an attribute, `Location`, representing the location of the gene on a chromosome. Let *brca* be an object of `GENE`: gene *brca* has two versions, v and v', where v' is created in order to record a change of `Location` and is derived from v. A gene that was initially considered a distinct object can subsequently turn out to be a conflation of two genes. For example, suppose that it is discovered that *brca* represents in fact two genes, *brca1* and *brca2*. This can be represented in the database by splitting version v' of *brca* into two new versions of *brca1* and *brca2*, respectively. We propose version management facilities that allow modeling version merging and splitting across different objects.

The rest of the paper is organized as follows. Section 2 briefly describes an object data model that supports selective versioning (i.e., non-versioned and versioned attributes) as well as generic and specific version references. Version defaults are examined in section 3. Version management in class hierarchies is discussed in section 4. In section 5 we examine the version management facilities that support version derivations across different objects. Section 6 contains concluding remarks.

2 An Object Data Model with Versions

In this section, we review the main constructs of an object data model whose non-versioned part is closely related to other semantic [11, 12] and object data models [3]. The version constructs of this data model follow [19].

Objects in our object data model are uniquely identified by object identifiers (oids), are qualified by attributes, and are classified into classes. A subset of the attributes associated with a class is specified as the external object identifier (`ID`). A class can be defined as a subclass of other (super) classes, where a subclass *inherits* the attributes of its superclasses. Examples of object classes are shown in Fig. 1.

Attributes can be *simple* or consist of a *tuple* of simple attributes (e.g., `Location` of `GENE`). A simple attribute can have a single value, a set of values, or a list of values, and can be *primitive*, if it is associated with a system-provided data type, or *abstract*, if it takes values from object classes (e.g., `References` of `BOOK`).

The attributes of an object class can be partitioned into *non-versioned* and *versioned* attributes. Non-versioned attributes represent stable object properties (e.g., `SSN` of `PERSON`), while versioned attributes represent evolving object properties of an object (e.g., `Address` of `PERSON`). A versioned abstract attribute can have a *generic reference* to current (default) object versions (e.g., `References` of `BOOK`) or a *specific reference* to specific object versions (e.g., `Studies` of `SCIENTIST`).

The structure of non-versioned objects in an object class is defined below.

```
OBJECT CLASS PERSON
  ID: SSN
  ATTRIBUTE             SSN: CHAR(11)          not null
  ATTRIBUTE             Name: CHAR(50)
  VERSIONED  ATTRIBUTE Address: CHAR(150)
OBJECT CLASS SCIENTIST isa PERSON
  VERSIONED  ATTRIBUTE Rank: CHAR(50)
  VERSIONED  ATTRIBUTE Studies: set-of specific GENE
OBJECT CLASS GENE
  ID: Name
  ATTRIBUTE             Name: CHAR(10)         not null
  VERSIONED ATTRIBUTE  Location (Chromosome, Position): (CHROMOSOME,INT)
OBJECT CLASS CHROMOSOME
  ID: Number
  ATTRIBUTE             Number: CHAR(2)        not null
OBJECT CLASS BOOK
  ID: ISBN
  ATTRIBUTE             ISBN: CHAR(30)         not null
  ATTRIBUTE             Title: CHAR(150)
  VERSIONED  ATTRIBUTE References: set-of GENE
```

Fig. 1. Examples of Object Classes

Definition 1. Let O be an object class and x be an instance of O. Then $x = (oid(x), val(x))$, where $oid(x)$ is the globally unique object identifier of x and $val(x)$ is the value of x. $oid(x)$ takes values from a class of object identifiers. $val(x)$ has the form $(A_1 : A_1(x), \ldots, A_n : A_n(x))$, where A_j, $1 \leq j \leq n$, is the name of an attribute of O, and $A_j(x)$ denotes the value of attribute A_j for x and is defined as follows:

1. If A_j is a simple attribute then: (i) if A_j is a primitive attribute then $A_j(x)$ consists of a value, a set of values, or a list of values of the data type associated with A_j, or (ii) if A_j is an abstract attribute taking values from class O_k, then $A_j(x)$ consists of one element, a set of elements, or a list of elements from $\{ \ oid(y) \mid y \in O_k \ \}$.
2. If $A_j = (A_{j_1}, \ldots, A_{j_n})$, then $A_j(x)$ consists of a tuple, a set of tuples, or a list of tuples of the form $[A_{j_1}(x), \ldots, A_{j_n}(x)]$, where each component $A_{j_k}(x)$, $j_1 \leq j_k \leq j_n$, consists of an element of the form defined in 1 above.

A version of an object x consists of the values of the non-versioned attributes for x (common to all versions of x) and the values of versioned attributes for a specific version of x. An object with multiple versions has a *default* version which is selected from the set of versions for that object.

The structure of versioned objects in an object class is defined below.

Definition 2. Let O be a versioned object class and x be an instance of O. Then $x = (oid(x), val(x), \{ \ (vid(x_i), vval(x_i) \ \} \ )$, where $oid(x)$ is the globally unique object identifier of x, $val(x)$ is the non-versioned value of x, $vid(x_i)$ is a unique version identifier for version x_i of x, $vval(x_i)$ is a versioned value for version x_i of x. $oid(x)$ and $val(x)$ are defined as in Definition 1 above. $vid(x_i)$ takes values from a class of version identifiers. $vval(x_i)$ has the form $(A_1 : A_1(x_i)$, $\ldots$, $A_n : A_n(x_i))$, where A_j, $1 \leq j \leq n$, is the name of a versioned attribute of O, and $A_j(x_i)$ denotes the value of A_j for version x_i and is defined as follows:

1. If A_j is a simple versioned attribute then: (i) if A_j is a primitive attribute then $A_j(x)$ consists of a value, a set of values, or a list of values of the data type associated with A_j, or (ii) if A_j is an abstract attribute taking values from class O_k, then $A_j(x_i)$ consists of one element, a set of elements, or a list of elements from $\{\ (oid(y),\ vid(y_m)) \mid y \in O_k,\ y_m$ is a version of $y\ \}$.
2. If $A_j = (A_{j_1}, \ldots, A_{j_n})$, then $A_j(x_i)$ consists of a tuple, a set of tuples, or a list of tuples of the form $[A_{j_1}(x_i), \ldots, A_{j_n}(x_i)]$, where each component $A_{j_k}(x_i)$, $j_1 \leq j_k \leq j_n$, consists of an element of the form defined in 1 above.

A version x_v of x is defined as $(oid(x), val(x), vid(x_v), vval(x_v))$.

Version Management. Keeping track of object versions involves maintaining the following additional non-versioned system (i.e., system provided) attributes, for each object instance, x, where the values of these attributes are included in $val(x)$:

1. `defaultVersion` (x) consists of the identifier(s) of the default version(s) of x;
2. `deleteDate` (x) indicates the date/time x was marked as deleted.

The following versioned system attributes are also maintained for each version x_v of x, where the values of these attributes are included in $vval(x_v)$:

1. `fromDate` (x_v) indicates the date/time x_v was created;
2. `preVer` (x_v) consists of the identifier(s) of the predecessor version(s) of x_v;
3. `vdeleteDate` (x_v) indicates the date/time x_v was marked as deleted.

We assume that versioned objects and their versions are not physically deleted from the database, but only marked as (logically) deleted.

Version management data change as a result of data manipulations, such as creating a new version, editing or deleting an object, etc, or version operations such as setting a version default. The operations needed for version management include (t_{now} denotes the current date/time):

1. x = `new`() for creating a new versioned object, x, together with its first version, x_0; $val(x)$ includes: `defaultVersion`$(x) = vid(x_0)$, `deleteDate`(x) = null; $vval(x_0)$ includes `fromDate`$(x_0) = t_{now}$, `preVer`(x_0) = null, `vdeleteDate`(x_0) = null.
2. x_v = `new`$(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$, for creating a new version of object x, x_v, from one ($m = 1$) or several ($m > 1$) versions of x; $val(x)$ does not change; $vval(x_v)$ includes `fromDate`$(x_v) = t_{now}$, `vdeleteDate`(x_v) = null, `preVer`(x_v) consists of $vid(x_{i_1}), \ldots, vid(x_{i_m})$.
3. `ldelete_object`(x) for logically deleting instance x; `deleteDate`(x) and `vdeleteDate`(x_i) for all versions x_i of x are set to t_{now}.

4. `set_default`(x, x_v) for setting version x_v as the default version of x; `defaultVersion`(x) is set to $vid(x_v)$.

In addition, version management is involved in retrieval operations in a versioned database. We consider in this paper two such operations:

1. `get_all`($oid(x)$) for retrieving all versions of object x.
2. `get`($oid(x)$ [,$vid(x_v)$]) for retrieving version x_v of object x; if $vid(x_v)$ is not specified, then the default version of x is retrieved.

3 Default Versions

We consider in this paper versioned databases that require maintaining a lossless and consistent object history. We assume that all objects have at least one (initial) version, and that versioned objects and their versions are not physically removed (deleted) from the database. Object history losslessness guarantees recoverability of past states, while its consistency regards the validity of object references in each (i.e., past or current) database state. Thus, a database state at a given time t involves only references between objects and object versions that are not logically deleted at t; however, logical deletion of objects and object versions preserves past references to these objects and object versions.

The state of a versioned database is defined below.

Definition 3. A database state at (past or current) time t, $DS(t)$, consists of all objects, x, that are not logically deleted at t, (i.e., `deleteDate`(x) $> t$ or `deleteDate`(x) is null) and their versions, x_i, such that:

1. x_i is not logically deleted at t, that is, `vdeleteDate`(x_i) $> t$ or `vdeleteDate`(x_i) is null;
2. x_i has been created before or at t, that is, `fromDate`(x_i) $\leq t$; and
3. the attribute values for x_i are specified as specified in Definition 2 of section 2, where the value of a versioned abstract attribute of x, A, with generic reference to default (current) versions of objects in class O, consists of the version identifiers of the default versions of O objects at time t.

We examine in the rest of this section the problem of determining the default object versions at a given time t in a versioned database.

In a versioned database, usually one or more versions are selected from the set of versions of an object as its *default* versions. For simplicity, we assume below that at any given time t, an object can have only one default version. The default version of an object x is not necessarily the last one created, and it can be changed by setting another version as the new default for x.

Generic references to a versioned object automatically refer to its default version and are not invalidated when the referenced object or its default version are modified. Thus, the versions of an object x referencing the default version of object y are not affected by changes of y, including the replacement or deletion of the default of y. It is important to note, however, that the value of a versioned attribute with a generic reference depends on the time the value is evaluated. Consider two instances of object classes `BOOK` and `GENE`, as shown in Fig. 2, where `BOOK` has versioned attribute `References` referencing default (current) versions of `GENE`. For example, at time t_4 version x_2 of instance x of `BOOK` references default version y_4 of instance y of `GENE` (see Fig. 2). At a previous time, t (e.g., between t_0 and t_2), another version of x, x_0, was referencing the default version(s) of y at t rather than referencing the default version of y at t_4. Thus, at time t_1, x_0 could have referenced version y_0, y_1, or y_2 of y, depending on which one of these versions was the default version of y at t_1. Consequently, the history of default versions for each object needs to be maintained in order to infer the database state at any given time. If only one default version is allowed for each versioned object at any given time (as it is usually done), the default version history is linear (i.e., totally ordered), as shown in Fig. 2.

Default Version Management. Keeping track of the history of default versions for an object x requires maintaining an additional non-versioned tuple system attribute, `defaultHistory` [`defaultVer`, `defaultStart`, `defaultEnd`], where:

1. `defaultVer` (x) identifies a (current or old) default version of x, x_v;
2. `defaultStart` (x_v) indicates the date/time version x_v was selected as default version;
3. `defaultEnd` (x_v) indicates the date/time version x_v ceased to be a default version.

For example, the default history for `GENE` y is characterized as follows:

`defaultVer` y_0: `defaultStart` $= t_0$, `defaultEnd` $= t_1$;
`defaultVer` y_2: `defaultStart` $= t_1$, `defaultEnd` $= t_3$;
`defaultVer` y_3: `defaultStart` $= t_3$, `defaultEnd` $= t_4$;
`defaultVer` y_4: `defaultStart` $= t_4$, `defaultEnd` $=$ null.

With the history of default versions properly maintained, a versioned database state at any given time t can be easily determined. The default version for an

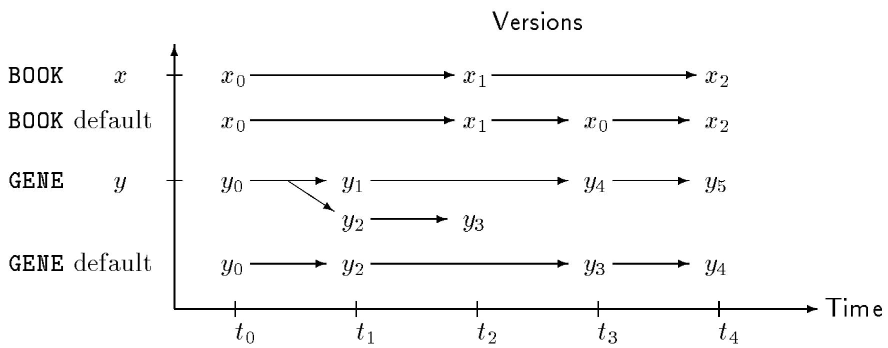

Fig. 2. Default Versions

object x at time t is the version x_v in the default history of x that has a pair ($\texttt{defaultStart}(x_v)$, $\texttt{defaultEnd}(x_v)$) satisfying the following condition:

$\texttt{defaultStart}(x_v) \leq t$ and ($\texttt{defaultEnd}(x_v) =$ null or $\texttt{defaultEnd}(x_v) > t$).

For example, for the objects and versions shown in Fig. 2, database state $DS(t_1)$ consists of instance x of `BOOK` with version x_0, and instance y of `GENE` with versions y_0, y_1 and y_2, where version x_0 of x references version y_2 of y, the default version of y at t_1.

4 Versions in Class Hierarchies

We examine in this section version management for objects belonging to multiple classes of a class hierarchy.

Consider an object class, O_i, and let x be an object of O_i. Suppose that changes to attribute values of x, where these attributes are associated directly with (i.e., are not inherited from) O_i, lead to the creation of a new version of x only in O_i, even if x belongs to other classes (e.g., a subclass of O_i). The following example shows that such a versioning mechanism is not sufficient for determining the state of x at a given time, t, that is, the values of all the attributes of x, including its inherited attributes, at t.

Example 1. (Fig. 3.) Consider class `PERSON` with versioned attribute `Address`, and subclass `SCIENTIST` of `PERSON`, with versioned attributes `Rank` and `Studies`.

Suppose that initially (i.e., at time t_0) instance x (version x'_0) is a `PERSON` object but not a `SCIENTIST` object. Subsequently (at time $t_1 > t_0$), x becomes a `SCIENTIST` object (version x''_0). Note that x'_0 and x''_0 have different creation times and involve different attribute values of x.

Later (at time $t_2 > t_1$), the `Studies` value of x changes and therefore a new version (x''_1) of x in `SCIENTIST` is created, but without creating a new version of x in `PERSON`. Then (at $t_3 > t_2$), the `Address` value of x changes and therefore a new version (x'_1) of x in `PERSON` is created, without creating a new version of x in `SCIENTIST`. Finally (at $t_4 > t_3$), the `Rank` value of x changes and therefore a new version (x''_2) of x in `SCIENTIST` is created.

Assuming that versions of an object x belonging to a class O_i involve only the values of the attributes associated directly with (i.e., not inherited by) O_i, if x belongs to multiple classes, O_i, $1 \leq i \leq m$, of a class hierarchy, then the state

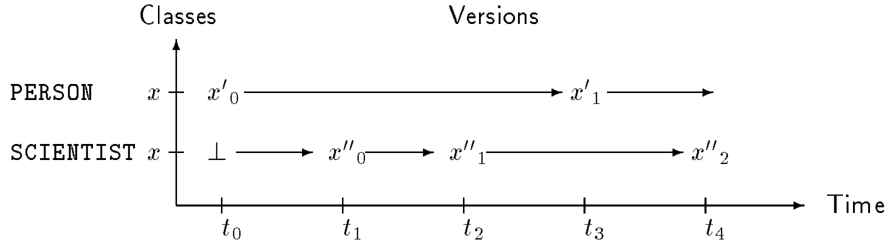

Fig. 3. Local Versions in Example 1

of x must be characterized by its versions in these classes. Note that version management data for versions of x in different classes must be maintained separately. However, some version management data (e.g., `defaultVersion`) must be synchronized across the classes of a class hierarchy. For the versions shown in Fig. 3, for example, it may not make sense to set version x'_1 as the default of `PERSON` x and version x''_0 as the default of `SCIENTIST` x.

Global Versions. An object belonging to multiple classes of a class hierarchy can be viewed as having *global* versions that involve all the attributes associated with these classes, where a global version of an object x is defined as the association of x versions that are local to the classes containing x. Note that a local version of x can be involved in several global versions of x.

Example 2. (Fig. 4.) Consider classes `PERSON` and `SCIENTIST` from the previous example. Suppose that initially (i.e., at time t_0) x is a `PERSON` (global version x_0, local version x'_0), but not a `SCIENTIST` object. Later, at time t_1, x becomes a `SCIENTIST` object (local version x''_0). A new global version, x_1, is defined as the association of local versions x'_0 and x''_0.

Subsequently, at time t_2, the `Studies` value of x changes and therefore a new local version, x''_1, of x in `SCIENTIST` is created; a new global version, x_2, is defined as the association of x'_0 and x''_1. At t_3, the `Address` value of x changes and therefore a new local version x'_1 of x in `PERSON` is created; global version x_3 is defined as the association of x'_1 and x''_1. Finally, at t_4, the `Rank` value of x changes and therefore a new local version, x''_2, of x in `SCIENTIST` is created; global version x_4 is defined as the association of x'_1 and x''_2.

Version Management. Version management data, such as the values of system attributes `defaultVersion` and `preVer`, refer to global versions. Similarly, version operations such as version creation and retrieval, refer to global versions.

Let x be an object belonging to multiple classes of a class hierarchy. A version of x local to a class in this hierarchy, x_i, continues to be identified by a unique version identifier, $vid(x_i)$ (see Definition 2 of section 2), while a global version of x, x_k, is identified by a global version identifier, $gvid(x_k)$.

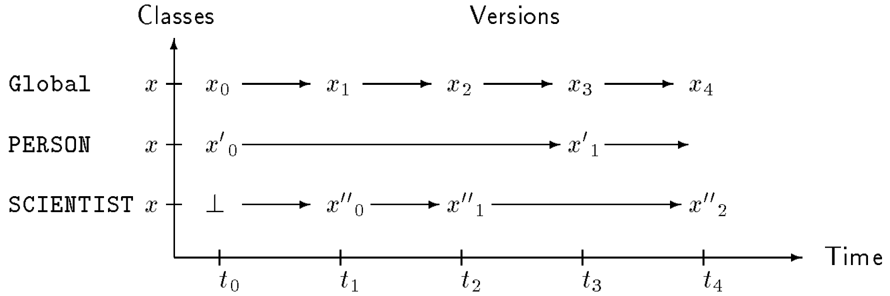

Fig. 4. Global and Local Versions in Example 2

The following additional version data are kept for each global version x_k of x: $\{ (gvid(x_k), lvid(x_k)) \}$, where $gvid(x_k)$ identifies global version x_k and $lvid(x_k)$ consists of the set of identifiers for the local versions associated with x_k. A global version of an object x is constructed by combining its associated local versions, as discussed below.

Creating, editing, deleting, and retrieving global versions involve creating, editing, deleting, and retrieving one or several local versions. Thus, retrieving (the attribute values of) a global version x_k of an object x, involves retrieving the local versions whose identifiers are associated with $gvid(x_k)$. For example, suppose that x_4 is the default global version of x in example 2 (see Fig. 4). Then $\texttt{get}(oid(x))$, which retrieves the default global version of x, consists of retrieving version x'_1 of `PERSON` and version x''_2 of `SCIENTIST` and then constructing x_4 from the attribute values of x'_1 and x''_2.

Alternative ways of maintaining global versions of an object x include:

(i) Keeping different identifiers for local and global versions, so that for a global version x_k associated with local versions $x_{i1}, \ldots, x_{im}$, $gvid(x_k)$ and $vid(x_{ij})$, $1 \leq j \leq m$, are distinct. This alternative requires maintaining global version identifiers and their association with local version identifiers, but limits local version creation to the classes associated with the attributes affected by a change.

(ii) Using common identifiers for local and global versions, so that $vid(x_i)$ would identify both local version x_i and the global version associated with x_i. This alternative involves a simpler version maintenance (and simpler version operations), but may entail data duplication. For an object instance x, for example, changes of the versioned attributes of x in class O_i, entail creating a new global version of x, x_n, identified by $vid(x_n)$, a new local version of x in O_i as well as new local versions in each class containing x, all identified by $vid(x_n)$.

5 Extended Version Derivations

Traditionally, different objects in a versioned database, that is, objects with distinct identifiers, are considered to represent different entities in the underlying application. In such a database, versions can be derived only from other versions of the same object and can be tracked back to one initial version of that object. If the version derivation for each object is represented by a directed acyclic graph (DAG),[3] then such DAGs have single roots representing the first version of each object, and DAGs representing versions of different objects are disconnected.

Traditional version derivation is not adequate to model the evolution of scientific database objects that often represent imperfectly known real-world entities.

[3] The nodes in the DAG represent versions and directed edges represent the derivation of the versions represented by their end nodes, from the versions represented by their start nodes.

In molecular biology, for example, it is not uncommon that two genes identified by distinct methods, and therefore represented by different objects in the database, are eventually found to be identical; or that experimental results indicate that what has been considered to be a single gene, represented by a single object in the database, is in fact a conflation of two genes. Modeling the evolution of scientific database objects representing such entities requires an extended version derivation.

Extended Version Derivations. The purpose of extended version derivations is to allow (i) creating (deriving) new objects in a class from versions of other objects in the same class, and (ii) setting versions of certain objects as defaults of the objects they have been derived from. Accordingly, the version management data associated with each object x of a versioned class O is modified as follows:

1. `preVer` (x_0) (predecessor of the initial version of x) can be either null or can consist of one or several pairs of the form $(oid(y), vid(y_i))$, where for each y, y belongs to O, y_i is a version of y, and $oid(y) \neq oid(x)$;
2. `defaultVersion` (x) (the default version for x) can either be null or consist of one or several pairs of the form $(oid(y), vid(y_i))$ or $(oid(y), *)$, where for each y, y belongs to O, y_i is a version of y, and '$*$' stands for `defaultVersion`(y).

Version Management. In order to support extended version derivation, the operations for creating new objects and for setting defaults (see section 2) must be extended as follows;

1. $x =$ `new`$([y_i, \ldots, z_j])$ creates a new versioned object of class O, x, together with its first version, x_0; y_i, …, z_j, represent versions of other objects of O, y, …, z; $val(x)$ includes: `defaultVersion`$(x) = vid(x_0)$, and `deleteDate`(x) = null; $vval(x_0)$ includes: `fromDate`$(x_0) = t_{now}$, `vdeleteDate` (x_0) = null, and `preVer`(x_0) which is either null or (if y_i, …, z_j, are specified) consists of $(oid(y), vid(y_i))$, …, $(oid(z), vid(z_j))$. Attribute values for x can be selected from the corresponding attribute values of $y_i, \ldots, z_j$, or can be assigned new values.
2. `set_default`(x, y, y_i) sets the version identified by the pair $(oid(y), vid(y_i))$ as the default version of x, where either $oid(y) = oid(x)$ or `preVer` (y_0) contains $(oid(x), vid(x_k))$, and x_k is a version of x;
   `set_default`$(x, y, *)$ sets the default version of y as the default version of x (so that whenever the default version of y is changed, the new default version of y becomes the default version of x), where `preVer` (y_0) contains $(oid(x), vid(x_k))$, and x_k is a version of x.

If an object y in class O is derived from another object of O, x, then specific references to (old) versions of x are allowed. However, whether generic references to current default versions of x are allowed or not can be application specific, and include the the following alternatives:

(i) x is not affected by the creation of a new object y from x, and therefore generic references to default (current) versions of x are allowed; these generic references refer to versions of x identified by the values of `defaultVersion`(x);

(ii) after the creation of a new object y from x, x is not considered valid anymore, and therefore references to default (current) versions of x are disallowed; or
(iii) the creation of a new object y from x is considered to represent the replacement of x by y, and therefore generic references to default versions of x automatically refer to the default version of y.

Alternative (i) is straightforward. Alternative (ii) can be carried out by setting `defaultVersion`(x) to null. Alternative (iii) can be carried out by setting `defaultVersion`(x) to $(oid(y), *)$, by `set_default`$(x, y, *)$.

Operations `get_all` and `get` will take into account the extended version management data. For example, suppose that object x has versions x_0, x_1 and x_2, and that at time t a new object y (version y_0) is derived from x_2. If alternative (ii) above is followed, then `get_all`$(oid(x))$ returns x_0, x_1 and x_2, while `get`$(oid(x))$ returns null. If alternative (iii) above is followed, then `get_all`$(oid(x))$ returns x_0, x_1, x_2 and y_0, while `get`$(oid(x))$ returns y_0.

Object Merging and Splitting. Version derivations spanning different objects allow modeling new operations based on objects, such as the object merging and object splitting discussed below.

Consider two instances of class `GENE`, instance x with versions x_0 and x_1, and instance y with versions y_0, y_1, and y_2, as shown in Fig. 5. Suppose that eventually instances x and y are found to represent the same gene. Then objects x and y can be *merged* into a new object, z, of `GENE`, by creating its initial version (z_0) from version x_1 of x and version y_2 of y. Note that version z_0 can be specified with non-versioned attribute values that are different from the non-versioned attribute values of x and y, can preserve (some of) the non-versioned attribute values of x or y. Note also that specifying a new version in this example involves specifying attribute values for the new version by possibly selecting versioned attribute values from x or y, or by assigning new values to the versioned attributes. A special case of object merging consists of creating a new version of an existing object, x, from another version of x and versions of other objects, such as y: this represents merging objects such as y, into x.

Similarly, consider an instance of class `GENE`, x, with versions x_0 and x_1, as shown in Fig. 6, and suppose that it is discovered subsequently that in fact x

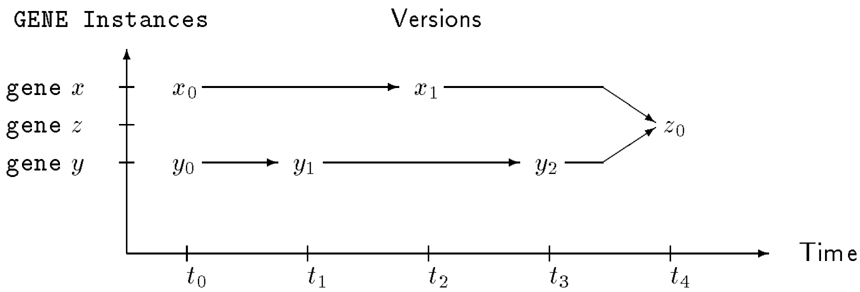

Fig. 5. Example of Merging Versions of Different Objects

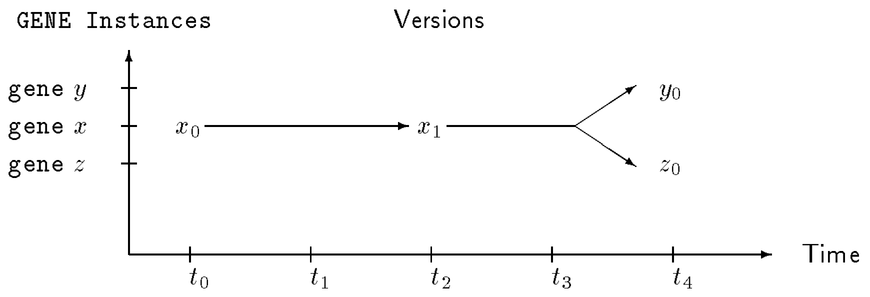

Fig. 6. Example of Splitting a Version into Versions of New Objects

represents a conflation of two genes. Then, x can be split into two new instances of class `GENE`, y and z, by creating these instances from version x_1 of x (see Fig. 6). A special case of version splitting is object cloning, where a version of a single new object, y, is created from a version of an existing object x.

Object merging and splitting can be used to correct the database representation of entities that cease to be current in the underlying application and therefore are superseded by other (current) entities. Object merging and splitting can be achieved using the extended `new` operation discussed above. The default of the merged or split object can be set as the default of the new objects created by merging or splitting, so that generic references to objects representing past objects will automatically point to current (default) versions of objects representing current objects.

For example, in Fig. 6, both y_0 and z_0 are versions of new objects created (derived) from version x_1 of object x. The default of object x, can be set to null, or can consist of the default version of y, the default version of z, or both. For version merging, the default of the merged objects (e.g., x and y in the example above) can be set to null or to the default of the object created by merging, z.

6 Concluding Remarks

While addressing general object version management problems, the facilities discussed in this paper were developed in order to support versioning in scientific (e.g., molecular biology, epidemiology) databases.

Part of the version management facilities presented in this paper have been incorporated into an object data model underlying a suite of data management tools used for developing scientific databases [5]. Since numerous scientific databases[4] are developed with commercial relational database management sys-

[4] For example, the Genome Data Base (GDB) at the Johns Hopkins School of Medicine, the Genome Sequence Data Base (GSDB) at the National Center for Genome Resources, the Protein Data Bank (PDB) at Brookhaven National Laboratory, Flybase at Harvard University, and the Mouse Genome Database (MGD) at Jackson Laboratory.

tems (DBMSs),[5] these facilities were implemented using relational constructs. Scientific databases often contain large amounts of data, therefore object versioning can lead to a substantial increase in their size and thus degrade their performance. We are currently experimenting with alternative relational implementations of the version management facilities discussed in this paper. where the overhead entailed by version management cannot be allowed to degrade the performance of the underlying databases.

Acknowledgements. The work of I-Min A. Chen and Victor M. Markowitz was supported by the Office of Health and Environmental Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098. The work of Stanley Letovsky, Peter Li, and Kenneth H. Fasman was supported in part by DOE award number DE-FC02-91ER61230.

We want to thank the anonymous referees for their valuable suggestions.

References

1. Ahmed, R., Navathe, S.B. Version Management of Composite Objects in CAD Databases. *Proceedings of the ACM SIGMOD Conference* (1991), 218–227.
2. Banerjee, J. et al. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems* **5**, 1 (January 1987), 3–26.
3. Bancilhon, F., Delobel. C., Kanellakis, P. Building an Object-Oriented Database System: The Story of O_2. Morgan Kaufmann Publishers, Inc. (1992).
4. Beech, D., Mahbod, B. Generalized Version Control in an Object-Oriented Database. *Proceedings of the 5th International Conference on Data Engineering* (1988), 14–22.
5. Chen, I.A., Markowitz, V.M.: An Overview of the Object-Protocol Model (OPM) and the OPM Data Management Tools. Information Systems **20**, 5 (1995) 393–417. Additional information available on the World Wide Web at http://gizmo.lbl.gov/opm.html.
6. Chou, H.-T., Kim, W. A Unified Framework for Version Control in a CAD Environment. *Proceedings of the 12th International Conference on Very Large Data Bases* (1986), 336–344.
7. Dadam, P., Lum, V., Werner, H.-D. Integration of Time Versions into a Relational Database System. *Proceedings of the 10th International Conference on Very Large Data Bases* (1984), 509–522.
8. Date, C.J. Referential Integrity, in *Relational Database – Selected Writings.* Addison-Wesley (1986).
9. Dittrich, K.R., Lorie, R.A. Version Support for Engineering Database Systems. *IEEE Trans. on Software Engineering* **14**, 4 (April 1988), 429–437.
10. Goodman, N. An Object-Oriented DBMS War Story: Developing a Genome Mapping Database in C++. In W. Kim (ed.), *Modern Database Management: Object-Oriented and Multidatabase Techniques*, ACM Press (1994), 216–237.

[5] For large archival scientific databases, commercial relational DBMSs are often considered more robust and efficient than object-oriented and object-relational DBMSs. See [10] for an account of problems encountered developing a scientific database with an object-oriented DBMS.

11. Hammer, M., McLeod, D. Database Description with SDM: A Semantic Database Model. *ACM Trans. on Database Systems* **6**, 3, (September 1981), 351–386.
12. Hull, R., King, R. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys* **19**, 3 (September 1987), 201–260.
13. Käfer, W., Schöning, H. Mapping a Version Model to a Complex-Object Data Model. *Proceedings of the 8th International Conference on Data Engineering* (1992), 348–357.
14. Katz, R.H., Chang, E., Bhateja, R. Version Modeling Concepts for Computer-Aided Design Databases. *Proceedings of the ACM SIGMOD Conference* (1986), 379–386.
15. Katz, R.H., Chang, E. Managing Change in a Computer-Aided Design Database. *Proceedings of the 13th International Conference on Very Large Data Bases* (1987), 455–462.
16. Katz, R.H. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys* **22**, 4 (December 1990), 375–408.
17. Kim, W., et al. Composite Object Support in an Object-Oriented Database System. *Proceedings of the OOPSLA'87 Conference* (1987).
18. Klahold, P., Schlageter, G., and Wilkes, W. A General Model for Version Management in Databases. *Proceedings of the 12th International Conference on Very Large Data Bases* (1986), 319–327.
19. Sciore, E. Versioning and Configuration Management in an Object-Oriented Data Model. *VLDB Journal* **3** (1994), 77–106.
20. Woelk, D., Kim, W., Luther, W. An Object-Oriented Approach to Multimedia Databases. *Proceedings of the ACM SIGMOD Conference* (1986), 311–325.
21. Woelk, D., Kim, W. Multimedia Information Management in an Object-Oriented Database System. *Proceedings of the 13th International Conference on Very Large Data Bases* (1987), 319–329.
22. Zdonik, S.B. Version Management in an Object-Oriented Database. *Proceedings of the International Workshop on Advanced Programming Environments* (1986), 405–422.