On Object and Database Versioning in Distributed Environment

Stéphane Gançarski[1], Jarogniew Rykowski[2] and Waldemar Wieczerzycki[2]

[1] LAMSADE - Université Paris Dauphine
Pl. du Mal de Lattre de Tassigny 75775 PARIS cedex 16
tél: +33 - 1 - 44 05 44 18 fax: +33 - 1 - 44 05 40 91
e-mail : gancarski@lamsade.dauphine.fr

[2] EFP - Ecole Franco-Polonaise en Nouvelles Technologies
de l'Information et de la Télécommunication
Mansfelda 4, 60-854 Poznañ, Poland
tel: +48 - 61 - 48 34 06 fax: +48 - 61 - 48 35 82
e-mail : {rykowski, wiecz}@efp.poznan.pl

Abstract

The paper presents how multiversion databases may be efficiently distributed in the case of most common multiversion applications.

First, the versioning model is described which allows the management of both object versions and configurations. This model, together with the data structures it uses, allows to manage efficiently as many versions as needed, and appears to be well suited for being adapted to distributed environment. Next, different approaches to database distribution are proposed which aim to reduce useless physical redundancy of object versions and minimize network transmission. These solutions take into account the nature and needs of applications of different kind. Finally, a study on implementation issues for each proposed solution is given. In particular, data structures are revisited to efficiently support the distribution.

Keywords: database distribution, version management

1. Introduction

Version control is an important functionality in a growing number of software applications. Very briefly speaking, version control is the ability to manage dependencies between subsequent instances of the same artifact (object), organize them into meaningful structures (e.g. sequences, trees or DAGs) reflecting the way versions are created (by derivation or merging), and allow such operations as navigation or computation on them. The most common and relatively easy to understand examples are time-based and author-based versioning which allow the coexistence, display and comparison of different versions of the same artifact, even when created at different times and/or by different authors.

Version management has long been recognized as a critical issue for some inherently cooperative applications like computer aided software engineering (CASE), design (CAD), configuration management, and authoring [CK86][EC94][KC87][KCB86]. In these areas, the important issue for versioning is to support opportunistic open-ended design tasks requiring the representation and preservation of historical perspectives in the design process, the reuse of previous designs, and the exploitation of alternative designs. Flexible comparing and merging tools support the integration of these alternative designs.

Another promising application of versioning is its use in providing structured but relaxed consistency control in widely distributed and open collaborative systems. Particularly in large scale environments, where the long time delays in network communications limit the usefulness of traditional concurrency control concepts like serializability, versioning provides an alternative mechanism for managing concurrent changes [CM86].

Intuitively, a multiversion database is a persistent object storing system in which every object may be potentially represented by practically unlimited number of its versions. In comparison to conventional (i.e. monoversion) databases, a multiversion database adds a new granule of data storage, namely an object version, which is somehow embedded in a corresponding multiversion object. Moreover, a new semantic relationship between versions of the same object is kept in the multiversion database, respectively for every database object.

There is also another important difference between monoversion and multiversion database which relates to the problem of finding subsets of database objects that "go together". As it is well known, whenever a transaction is addressed to the monoversion database, the database is assumed to be in a consistent state, i.e. in a state that corresponds to one of valid states of the real-world being modeled in the database. Contrarily to monoversion database, a multiversion database is in general not consistent according to the classical definition of the consistency. More precisely, considered as a whole, it does not correspond to any valid state of the real-world. That is why, one must distinguish particular subsets of object versions that are mutually consistent in order to address to them users' transactions, which are assumed to preserve the consistency of those subsets after transactions commitment. Relatively to this issue, existing version models may be divided into three families, according to the granularity of the unit of versioning.

In the first family [AN91], [KS92], [TOC93], a multiversion object is a versioning granule. This implies that to associate consistent object versions, it is necessary to explicitly link versions: object versions may be referenced by other versions to build complex object versions. This link is used once as a reference of an entity to another, and once to associate two mutually consistent entity versions. This duality generates problems to manage consistent object version sets: creation of object versions implies creation of a new complex object version, mainly achieved manually or by automatic generation of many, often useless, complex object versions (percolation [Atw86]). Thus, those works are limited both by the number of complex object versions managed simultaneously and by the functionalities to manipulate them.

In the second family of version models [KRW92], [EC94], generally devoted to design applications, a subset of objects, called *generic configuration*, is a unit of both versioning and consistency. This implies that a new object version should be created only inside a particular configuration (i.e. a version of a generic configuration), thus avoiding the problem of explicit linking of mutually consistent object versions. However, this approach is restrictive, since it usually imposes users to use a *top-down* design. Moreover, sharing objects between different generic configurations raise difficulties similar to those appearing in the first family.

Finally, in the third approach, the whole „monoversion” database is a versioning unit. Units of consistency across versions are thus composed of versions of all the objects in the database, one version per object. This approach, in some sense, is used for example in temporal databases [Sarda90][TCGJSS93] which represent different dimensions of time (e.g. validity time, transaction time etc.). If only one dimension of time is represented, for instance the validity time, the database stores as many states of the real world as there are different validity states present in the database. However, the total order of versions imposed by the semantics of time flow induces a strong limitation on applying this approach to other domains, like design applications (CAD, CASE), where versions are generally organized in a tree or a DAG, instead of a sequence. On the contrary, in the *database version model* [CJ90], the organization of versions coming from the derivation is not limited to a sequential order. This model is very straightforward and natural since it reflects both progressive and alternative nature of real-world processes being modeled in the database. These two important advantages have convinced us to use the database version model in the further discussion.

The number of object versions may be arbitrarily big. Thus, in most cases, the size of the entire multiversion database is much more greater than the size of a conventional database. In configuration management, where configurations are composed of thousands of objects, each of them existing sometimes in more than a hundred of versions, the size of the database forbids to store it on a conventional server. In such situation, the database management system requires usually many more resources (e.g. disks, streamers) than those available at a single computer site. Thus, the natural requirement is to distribute the database over a computer network and to use all network resources.

Another important argument for database distribution is that the users of a multiversion database, usually grouped in teams or classes, tend to access subsets of database objects related to the tasks they perform, rather than the whole database. For instance, in a CASE application, the development team is working on current versions of programs while hot-line is dealing with released configurations. Those subsets should be stored as close as possible (ideally on the same local site) to the respective users, while others may be stored in a geographically different place, which, however, is accessed via the network.

Object versions, of course, may not be distributed randomly, but in a way that guarantees efficient use of the database by all users connected to the network. In order to do this properly, one must take into account two opposite criteria, and try to find a sort of trade-off which satisfies most of the database users. First, the replication of database objects (object versions) on different computer sites should be avoided, whenever possible. Physical copies of the same object (object versions) increase the size of the whole distributed database, they burden the storage buffers and mechanisms, as well as they overload the system to maintain consistency across copies. Second, one must allow each site to reach all the data it needs using the less network transmissions as possible. For instance, when one wants to compile a program for testing a piece of code, he (she) requires not to wait for the system to get through the network all the subprograms, header files and libraries needed for the compilation.

Those two criteria have different weights depending on the application specificity. For instance, when objects are read much more often than updated, it might be better to keep copies on each site where they may be requested, in order to speed up accesses, since the system will not have to keep consistency across copies very often. However, in most cases, each data stored in the database is related to one particular computer site, which is usually responsible for it and accesses it much more often than other sites in the network. Taking into account this natural assumption, it is possible to maximize simultaneously both criteria, by distributing the multiversion database according to application nature.

In the following, we show how multiversion databases may be efficiently distributed in the case of most common multiversion applications. The main contribution of the paper comprises the following elements.

- The use of a versioning model which allows the management of both object versions and configurations. This model, together with the data structures it uses, allows to manage efficiently as many versions as needed, and appears to be well suited for being adapted to distributed environment.
- Solutions maximizing the criteria given above, i.e. reducing useless physical copies as well as useless network transmissions. Those solutions take into account the nature and needs of applications of different kind.
- A study on implementation issues for each proposed solution. In particular, data structures are revisited to cope with distribution.

It is worth to emphasize that we consider in this paper the term *distributed database* [OV91][BG92] in its more general sense, which makes it different from conventional databases, namely that "data are partitioned and stored at separate computers" [GMH95]. We address the main issues of distributed databases, i.e. data location and consistency preserving, but do not consider related potential of distributed databases, such as parallel query processing or improving reliability by data replication. Thus, in the following, we propose solutions that minimize useless data replication in order to preserve consistency at the lowest cost in terms of network transmission.

This paper is organized as follows. In Section 2 the database version model is briefly reminded. Section 3 proposes three different approaches for a multiversion database distribution, managed according to the database version model, in which different granules of distribution are used. In Section 4 the basic implementation issues, respectively for three different distribution approaches, are considered. Finally, Section 5 summarizes the paper.

2. Versioning Model

One of the main features of the *database version approach* (*DBV* approach), which distinguishes it from other approaches, is the distinction between the logical level, „what the user sees”, and the physical level, „what is managed by the system”. In the *DBV* approach, the classical concept of „version of object” is

separated in two new concepts: a *logical version* of an object represents the version of this object as it is seen by users in a given context; a *physical version* of an object is used by the system to implement logical versions of this object having the same value.

At the logical level, the multiversion database is seen as a set of database versions, identified by a database version identifier (*dbvid*). Each *DBV* represents a context, a possible state of the modeled world, and for instance a natural way to store *configurations*. It is composed of one logical version of each object, identified by a pair (object identifier, database version identifier). If object *o* does not exist in *DBV d*, the value of its logical version *(o,d)* is *nil*. *DBV*s are created by logical copy, or *derivation*, from an existing *DBV*. Thus, at the creation of the multiversion database, a *root DBV* is generated. Users always manipulate logical versions of objects within their context (the *DBV* they belong to), and a new object version cannot be created outside an existing *DBV*. Thus, database versions are both unit of versioning and unit of consistency for object versions. Two kinds of transaction are defined: transactions that update logical versions of objects, called *object transactions*, and transactions that create or delete *DBV*s, called *dbv-transactions*.

At the physical level, each multiversion object is composed of a set of its physical versions, identified by physical version identifiers (*pvid*), and of a table, called *association table*, mapping *dbvid* to *pvid*. This mapping allows to retrieve the physical version of an object, and thus the value associated with a given logical version of this object. When a *DBV d_2* is created by logical copy of *d_1, d_2* shares all the physical versions of object with *d_1*. To avoid the updating of the association table of all the multiversion objects, the system stores the derivation link between *DBV*s and uses the following rule:

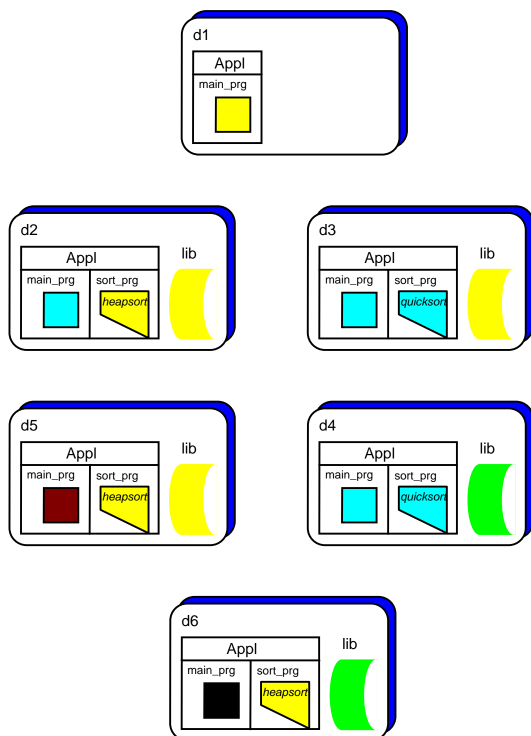
> the physical version associated with a given logical version *(o,d)* is the physical version associated either directly with *DBV d* by *(o,d)* in the association table of *o*, or with *(o,d')* in this table, *d'* being the nearest ascendant of *d*, relatively to the derivation link.

Fig. 1 gives a simplified example of a multiversion database for software development. The object *Appl* represents the application to be developed. It is a *complex object*, composed of programs. The object *main_prg* is the main program of the application, the object *sort_prg* is a sorting program, the object *lib* is a library containing sorting routines. The database is composed of six database versions: *d1, d2, d3, d4, d5, d6*, plus the root *DBV*. The derivation tree is shown at the bottom of the figure.

The left side of the figure presents the logical level of the database, i.e. the six database versions with their content, as they are seen by users (the root *DBV* is not represented because it is hidden to users). The object *lib* does not appear in *d1* because the value of its version in *DBV d1* is *nil*. As logical versions of the object *Appl* are *versions of a complex object* (i.e. they refer to other versions of objects), they are represented by frames, where the referred versions of objects are embedded. Logical versions of other objects are *versions of simple objects*, they do not refer to any version of object.

The right side of the figure presents the physical level of the database: the association table of each object and the derivation links between pairs of *DBV*s. For example, the physical version *s_v1* of object *sort_prg*, which uses an heapsort algorithm, is associated with *DBV*s *d2* and *d6*: the logical versions *(sort_prg, d2)* and *(sort_prg, d6)* are both associated with the physical version *s_v1* (i.e. they both use a heapsort). *d5*, derived from *d2*, does not appear in the association table of *sort_prg*: the logical version *(sort_prg, d5)* shares the physical version *s_v1* of *sort_prg* with *(sort_prg, d2).* Notice that the value of versions of complex object *Appl* contains object identifiers and not identifiers of versions of objects. For a logical version of *Appl* in a given *DBV d*, these object identifiers refer to the logical version of the specified objects in *DBV d*. For instance, as the value of *(Appl,d6)* contains the object identifiers *main.prg* and *sort.prg*, *(Appl,d6)* refers to the logical versions *(main.prg, d6)* and *(sort.prg,d6).* This implies that updating a component of a logical version of complex object does not affect the value of the other logical versions of this complex object, even if it is shared by some of them. In Fig. 1, *(Appl,d6)* and *(Appl,d4)* share the same version of component *main_prg*. If someone updates the main program of the application in *DBV d6*, a new physical version *m_v5* of the object *main* will be created and associated with *d6*. However, as for object *Appl*, *d4* remains associated with a value containing *m_v4, (Appl,d4)* still refers to the same version of the main program.

Logical level: 6 database versions and their content



Physical level

multiversion objects and their association tables

main_prg

pvid	dbvid
m_v0	root
m_v1	d1
m_v2	d2
m_v3	d5
m_v4	d6

sort_prg

pvid	dbvid
s_v0	root
s_v1	d2, d6
s_v2	d3

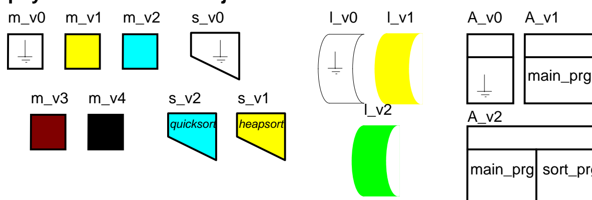
lib

pvid	dbvid
l_v0	root
l_v1	d2
l_v2	d3

Appl

pvid	dbvid
A_v0	root
A_v1	d1
A_v2	d2

physical versions of objects



derivation links between DBVs

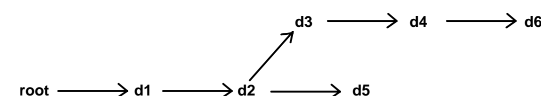


Figure 1. Logical and physical levels of multiversion database

Obviously, distributing a multiversion database, which is managed according to the DBV approach, must take into account the physical level of storing the multiversion data: physical object versions, association tables and the database version derivation tree. These data structures have to be organized and stored in such a way that object versions are efficiently available on any site they may be requested. Thus, the most important issue is to distribute object versions efficiently, according to applications needs. In such case association tables and derivation tree should be adapted to support distributed access to object versions. These two points are studied in details in the next two sections.

3. Granules of Distribution

Objects and data structures supporting database management may be distributed in the network in many ways. Some of them may be stored as a single copy kept on a particular network site, while other may be replicated many times and stored simultaneously on different sites. The way objects are distributed substantially impacts the efficiency of the distributed database.

In the three following subsections we propose three different approaches to object version distribution in the database managed according to the model presented in Section 2. These approaches differ by the elementary granule of distribution and are addressed to different areas of database application. Depending on the specificity of application, in particular, depending on the average number of versions of the same object accessed by the user, one of them might be more efficient and thus recommended, while the others - less efficient.

3.1. Horizontal Distribution

As it was mentioned (cf. Section 2), in the database version model a single database version is a unit of both versioning and consistency. Thus, the most natural and straightforward approach to object distribution is to treat a database version also as an elementary unit of distribution. This approach might become very useful in the case of applications in which the user tends to access exclusively his (her) private database version or a

set of private database versions, rather than public database versions, shared simultaneously by many users. Moreover, the user accesses most of objects stored in the database version addressed rather than a small subset only. This is very typical to *CASE* databases, when users work on parts of code assigned to them mostly separately, until they are ready for final code linking in a public database version. Even if a particular user works only on procedures (modules) assigned to him (her), he (she) still requires procedures being developed by other members of a team as well, or at least their signatures, in order to compile his (her) code and to test it. Versions of procedures developed by other users need not be the most recent at these stages of software development. It is because the user tests (debugs) the software only partially, putting a particular emphasis on going through all possible tracks in his (her) part of code, rather than in the whole program.

In situations like those mentioned above, database versions accessed by a single user should be entirely stored on his (her) computer site, in order to avoid object version transmission via the network. As a consequence, when a particular object is addressed in a user's database version, all other object versions, possibly bound to it by different semantic relationships, are also immediately available, because they belong to the same database version. It is illustrated in Fig. 2: the horizontal axis represents multiversion objects stored in the database, while the vertical axis represent database versions. Two database versions are stored on site *S1*, one on site *S2* and two on site *S3*. According to the database version model, a single version of every object kept in the database is available in every database version, in particular case - *nil* version.

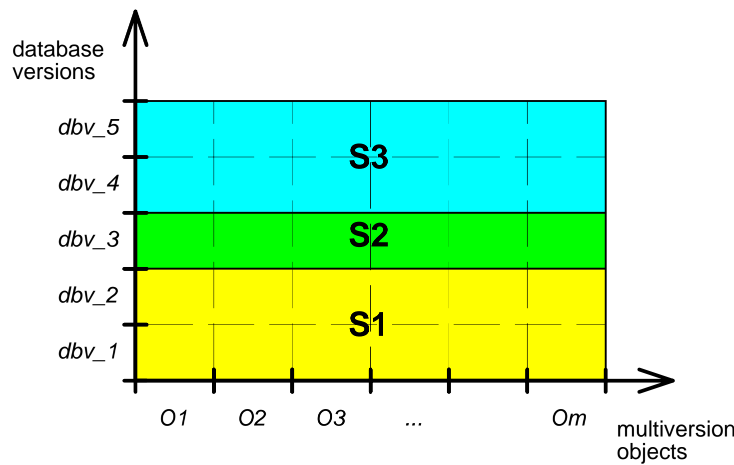


Figure 2. Horizontal distribution

Before discussing the advantages of the proposed approach to object version distribution, we give some observations concerning relationships among database versions stored on the same computer site, on one hand, and those stored on different sites, on the other hand.

Typically, the number of database versions in the multiversion database is much more greater than the number of computer sites, which means that in general on every computer site *n* database versions are stored. Moreover, database versions stored on the same site are typically directly bound by the derivation relationship, because they are operated by the same user. In other words, database versions stored on the same site typically belong to the same subtree of the derivation hierarchy. It is illustrated in Fig. 3. The correspondence between database versions and computer sites given in Fig. 3a is much more typical and probable than that in Fig. 3b.

Another observation concerns sharing of object versions between database versions. Typically the highest level of sharing appears between database versions directly bound by the derivation relationship, i.e. parent and child database versions, due to the nature of dbv-transactions. Transitively, there is usually also a relatively high level of object sharing between database versions indirectly bound by the derivation relationship, i.e. between database versions belonging to the same subtree of the derivation hierarchy. On the contrary, database versions included in different subtrees, e.g. subtrees stored on sites *S1* and *S3* in Fig. 3a, do not share object versions (except of *nil* version), or the sharing is very rare. As a consequence, in the approach proposed, object versions are mostly shared among database versions stored on the same computer site, while sharing between different sites is incidental.

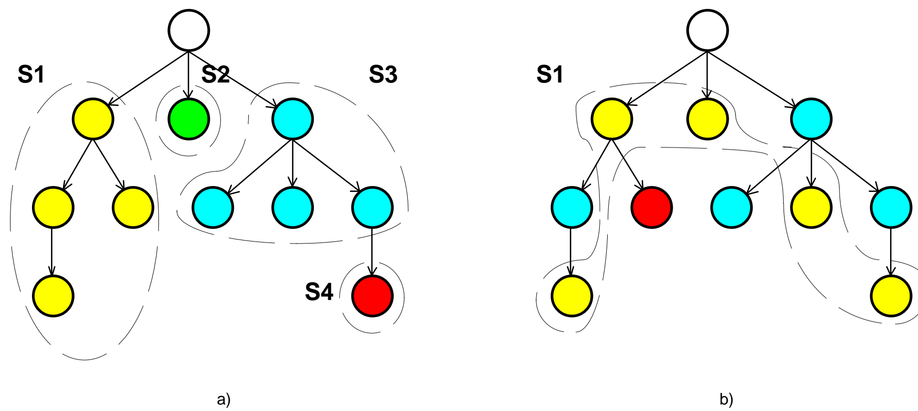


Figure 3. Correspondence between database versions and computer sites

As it is known (cf. Section 2), database versions are logically independent. Thus, an object version that appears simultaneously in two database versions stored on the same site is logically replicated, however physically there is no redundancy, and only one copy of shared object version is kept on the corresponding computer site. On the contrary, an object version that appears in two database versions stored on different computer sites is replicated both logically and physically. In other words, every site has its own physical copy of that object version. Those copies are independent, i.e. updating one of them does not impact the value of the second one. Due to the previous discussion, this replication does not substantially increase the level of redundancy in the distributed environment, providing that the assignment of database versions to computer sites corresponds to the one given in Fig. 3a.

The evident advantage of the proposed distribution technique is that every time the user addresses his (her) database versions, i.e. one of those stored on his (her) computer site, there is no need for network transmission, no matter if a given object is local or appears simultaneously in many database versions. In other words, every read, create or update operation may be performed immediately because a proper object version is always stored on the computer site the user is accessing. Notice, that even if the assignment of database versions to computer sites resembles the one presented in Fig. 3b, rather than in Fig. 3a, this feature is also preserved. In this case, however, the level of redundancy due to object version replication might be substantially higher.

Up till now, we have assumed that the user addresses a database version that is assigned to his (her) computer site. What will happen, however, if, for some reasons, he (she) decides to access a database version that is stored on a remote computer site? If the access request is incidental then an object version addressed may be transmitted over the network. Otherwise, i.e. if the user plans to access a particular remote database version more regularly, the following steps are performed.

First a new database version is created from scratch, i.e. a database version composed of *nil* object versions only, which in the global derivation hierarchy becomes a next child of a parent node of the subtree assigned to the computer site accessed by the user (or a selected subtree, if more than one subtrees are assigned). Next, the remote database version is entirely copied via the network to the one newly created, i.e. all not *nil* object versions belonging to the remote database version are physically replicated. This operation may cause some trouble due to object version transmission, proportionally to the number of objects existing in the remote database version. To avoid it so called *lazy replication* technique is applied, which is particularly beneficial if the user accesses only a subset of objects from the remote database version, and deletes the database version at the end of his (her) session.

The lazy replication technique consists in keeping in the database a particular link between local and remote database versions and delaying object version replication until they are explicitly addressed by the user. Because the remote database version may not be frozen, i.e. it may continue to evolve independently from the local database version, the link in fact is established between the local database version an a *shadow database version* of the remote one (both of them are stored on the same site). The shadow database is a sort of consistent snapshot made at the time user requested an access to the remote database version. The shadow database version is stored in the system until last not *nil* object version is replicated, or the local database version linked to it is deleted by the user.

3.2. Vertical Distribution

Another approach to object distribution is to treat a multiversion object as an elementary unit of distribution. This approach can be useful in a case of applications in which the user usually works with a small subset of objects from the database, and the versions of these objects are not shared with other users. This is a typical case of *CAD* databases in which, at early stages of the design process, users work independently on parts of the whole design artifact, assigned to them by the head of a design team. After many go-ahead and roll-back steps each of them prepares a version of the part he (she) is responsible, which is ripe enough to be released and shown to other users. Finally, in case of common acceptation, all parts are merged together in order to produce the artifact being an output of the design process.

In a situation like the one mentioned above, all versions of one multiversion object, created by a particular user, must be kept at one place, on his (her)computer site. It is illustrated in Fig. 4: the horizontal axis represents multiversion objects stored in the database, while the vertical axis represents database versions. Three multiversion objects are stored on site *S1*, one on site *S2* and two on site *S3*.

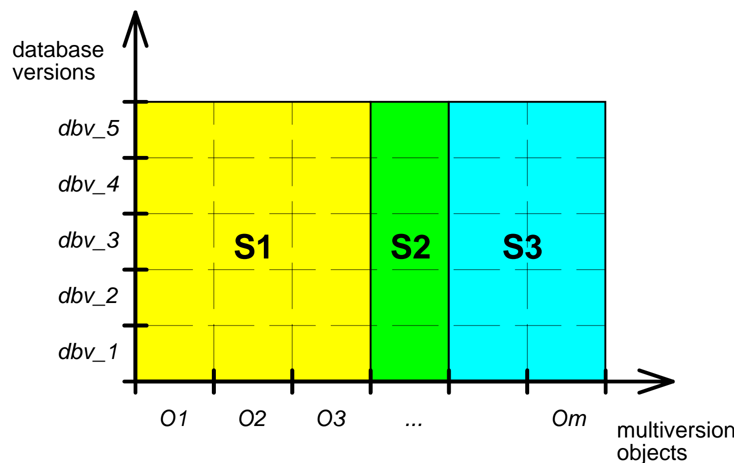


Figure 4. Vertical distribution

Now we discuss some observations concerning relationships among multiversion objects stored on the same computer site and on different sites.

Typically the number of multiversion objects stored in the multiversion database is much more greater than the number of computer sites, which means that in general on every computer site all versions of *n* multiversion objects are stored. Moreover, multiversion objects stored on the same site are typically directly bound by various semantic relationships, with the composition relationship being the most important one. It is worth to note that multiversion objects stored on the same site typically belong to the same subtree of the composition hierarchy. It is illustrated in Fig. 5. The correspondence between multiversion objects and computer sites given in Fig. 5a is much more typical and probable than that in Fig. 5b. This correspondence must be stored in a separate data structure, mapping object identifiers to computer site identifiers, in order to maintain well-known „data location transparency".

Another observation concerns physical sharing of object versions between database versions. Taking into account the fact that all the versions of a multiversion object are stored on the same site, we may say that the original advantage of physical sharing of value-equal versions (cf. Section 2) is preserved.

Up till now we have assumed that the user addresses multiversion objects which are assigned to his (her) computer site. If, however, he (she) has to access a version of a multiversion object that is stored on a remote computer site, two approaches can be used**.**

In the first approach, we assume that a user is not interested in keeping the consistency between local and remote copies of a multiversion object. For example, he (she) makes a „snapshot" of a remote object and does not care about its future evolution. This is typical to CAD and CASE systems, where some remote objects are read to perform a test, or to check the (local) integrity constraints. After test execution, the user does not need the objects anymore or at least until next test execution. To make a physical copy of a remote object, the user first creates a new local multiversion object from scratch, containing *nil* version only. Next, the given

multiversion object is copied from the remote site, i.e. all not *nil* object versions of the remote object are physically replicated. This operation may be very long, so to speed it up the lazy replication technique can be applied, in a way explained in Section 3.1.

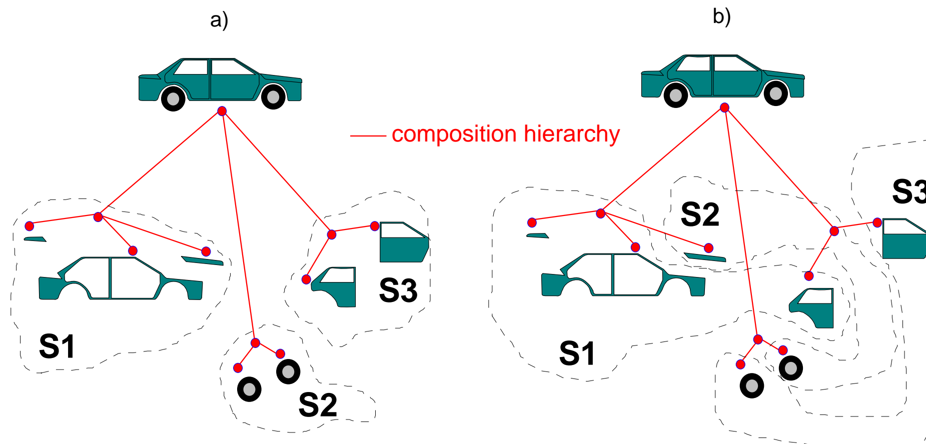


Figure 5. Correspondence between multiversion objects and computer sites

If the user is interested in keeping consistency between local and remote copies of the same object, he (she) may use a second approach, in which every update of a remote object copy is (immediately or in a lazy way) propagated to the local copy.

3.3. Mixed Distribution

As they are presented in the two previous subsections, horizontal and vertical approaches have different features. In the first case, object version replication and network transmission are avoided thanks to the particular semantics of dbv-transactions, assuming that database versions belonging to one site are bound by the derivation relationship. It is there well suited for applications where one user mostly access his (her) own database version but need to have a snapshot of almost every object. In the second case, the same result is obtained assuming that a given site is devoted to only some objects, but in all of their versions. This implies that two different sites may work concurrently on the same DBV, without too many remote access.

In some cases, these two features have to be both provided. Some sites are devoted to only *some versions of some objects*. This is typically the case of Configuration Management, when complex systems are developed, released, installed and maintained. Those complex systems are usually composed of both software and hardware, each of them being divided into a common part (e.g. operating system, basic resources), a country-related part (e.g. user interface messages, devices voltage) and a client-specific part (e.g. programs computing salaries according to client policy). Due to the complexity of the task, work is performed by different teams, working on different sites in different countries. For instance, one site in UK may be devoted to maintain versions of common hardware components, another site in France for the french and belgium versions of user interface, and so on.

In such application, it appears that neither whole database versions nor whole multiversion objects have to be fully stored on a given site. Actually, the elementary granule of distribution here is the single object version. In the general case, one may think about distributing object versions randomly, as it is suggested in Fig. 5: each site is assigned with a non structured set of object versions. From user point of view, this might be well suited, since he (she) can choose his (hers) own object versions without any restriction.

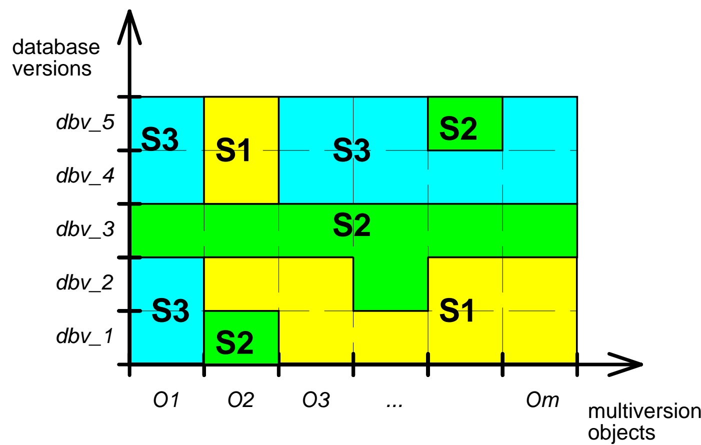


Figure 5. Random Mixed distribution

However, in this case, it is not possible to systematically avoid useless data replications or network transmissions. More precisely, since no assumption may be done on the database versions and on the object located in a given site, it is not possible to adapt vertical and horizontal solutions described before in a straightforward way. Thus, a more restricted but more realistic approach is the following, so called bounded mixed distribution. In the bounded mixed approach, each site is assigned some objects O1, O2,..Oi and some database versions dbv_1,..dbv_j, and is therefore assigned all the corresponding object versions, namely (O1,dbv_1), (O1,dbv_2) ... (Oi,dbv_j). In other words, a site is always represented in the two-dimension diagram, by a full rectangle, as it is represented on Fig. 6.

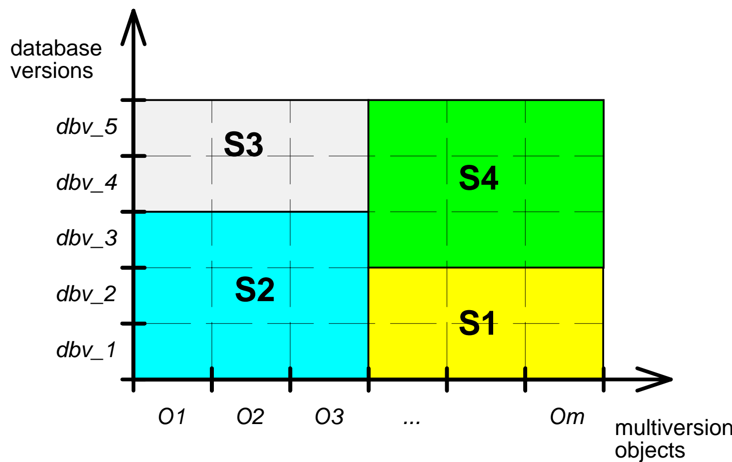


Figure 6. Bounded Mixed distribution

The main advantage of this solution is that it generalizes the two former. Since the granules of distribution may be chosen in a more flexible way, the trade off between object version replication and network transmission may be more efficiently found. Moreover, horizontal distribution and vertical distribution techniques may both be used, since here we may easily assume that [1] dbv_1,..dbv_j are included in a subtree of the derivation tree, and that [2] O1, O2,..Oi are semantically related objects. Then, we may assume that access to remote sites will occur very rarely, and that sharing of object versions occurs within sites much more than between sites.

Now consider the situation of remote accesses. Again, if an access is incidental, a message is send to the remote site and addressed object versions are transmitted over the network. For regular remote access, the situation is a little bit more complicated and requires to adapt previous solutions. Indeed, in order to allow regular access to remote sites, lazy replication must here be applied to „version rectangles", not to whole database versions nor to whole objects. Version rectangles are composed of a set of rows or by a set of columns located in the same site on the 2-D diagram. Assume for instance that a user working on site S1 decides to regularly access to versions of O1, O2, O3 in database versions dbv_3. Then, the corresponding rectangle must be replicated in site S1. In order to perform it in a lazy way, a consistent snapshot of the

rectangle has to stored in the remote site until last not nil object version it contains is effectively replicated. To perform this, one shadow database version and three shadow objects have to be created for the purpose of accessing only three object versions. This implies that the database might be soon crowded by shadow structures, however, as it is mentioned above, this situation is assume to occur very rarely.

Notice that, according to Fig. 5, another retriction may be put over mixed distribution, so called fixed object clustering. This means that, whenever a set of object is bound in a given site, no other set of object can overlap with it in another site. In the 2-D diagram, this means that sites are forming vertical layers of fixed size. This restriction simplify implementation as it will be discussed in Section 4.3. Moreover, objects are clustered according to composition links. In many cases, composition hierarchy is not evolving across versions, making fixed object clustering not too restrictive solution.

4. Implementation Issues

In this section we briefly discuss some implementation issues related to the distribution of object versions in the network, respectively for the three approaches (and the three different distribution granules) proposed in Section 3. The main contribution, however, concerns the implementation of the horizontal distribution (cf. Section 3.1). In case of two other approaches: the vertical and the mixed distribution, we show only the necessary modifications of the horizontal distribution implementation technique. An emphasis is put on the management of two main data structures, namely the database version tree and the association tables (cf. Section 2) in a distributed environment. Also, additional data structures, necessary to support an access to object versions stored in remote computer sites, are briefly mentioned.

4.1. Horizontal Distribution

Association tables may become very large due to practically unlimited number of object versions. They are also frequently updated as a consequence of updates of objects in different database versions. In general, association tables may be managed in two different ways. If versions of the same object tend to be grouped on one computer site and do not occur on other computer sites or occur rarely, then the computer site holding the majority of versions should also store the corresponding association table. The identifiers of object versions stored in different computer sites, which are included in association tables, must also be extended by computer site identifiers. In order to find the location of an association table stored out of a given computer site, the site must be also provided with an additional data structure mapping multiversion object identifiers into computer site identifiers.

Most accesses to an association table are related to operations on object versions stored on the same computer site as the corresponding association table. In this case, there is no overhead impacted by object distribution, i.e. all operations may be performed in the same way they are performed in the centralized environment. If, however, the required association table, or the requested object version, or both the association table and the object version are stored on different sites than the one accessed by the user, then exchange of messages among computer sites is necessary, i.e. the network has to be used. The site storing an association table is responsible for sending object version identifiers to other sites or updating the association table according to remote requests. The same concerns the site storing the addressed object version which must be transmitted to the respective remote site, and locked in the host site in a way avoiding access conflicts, until a corresponding remote transaction commits.

If versions of the same object are not grouped but distributed in the whole network, then the approach proposed above is not recommended. In such situation, instead of centralized association tables, distributed association tables should be used. If versions of a particular object are stored on *n* computer sites, then the association table should be divided into *n* parts, each one holding the rows describing object versions stored on the same computer site, and stored together with them (i.e. on the same site). Now, if the user accesses database version stored on his (her) computer site, no matter which object he (she) addresses, there is no need for network transmission: both a required part of an association table and object versions it refers are stored on the same site.

The above protocol becomes more complicated if the user accesses a remote database version, in which case a particular message exchange protocol must be applied, i.e. the network must be used. In case of many

accesses to a remote database version, to avoid data transmission, it is suggested to derive a new database version stored at the computer the user accesses and to link it with a remote one, as explained in Section 3.1.

The database version derivation tree is a data structure that does not require too much memory to be allocated, i.e. it may be easily encoded on few disk pages. It is also mostly read and updated only in case of dbv-transactions deriving new database versions. For this two reasons, it is replicated on every computer site of the distributed environment, and thus available immediately whenever required. In case of updates, the computer site that introduces them is responsible for informing all other computer sites and sending the updated derivation tree to all other sites.

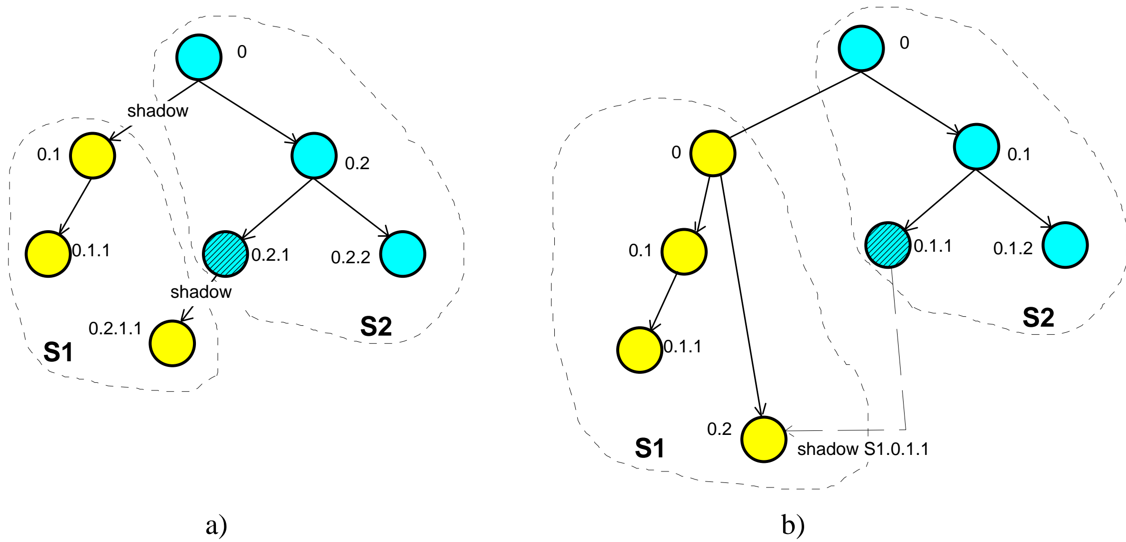
However, if the information about the derivation of all the database versions does not need to be known on all the sites, it is not reasonable to store database version derivation tree on a given site and to distribute it across the network. This tree should be rather distributed, according to the distribution of the database versions.

One may notice that all database versions are created either in a „shadow" way or as children of local database versions. The second case is rather obvious and does not require any further discussion. In the first case, a particular continuos link between a local database version and a corresponding remote database version may be created. Now, from the implementation point of view, it is sufficient to distinguish between local version stamps (beginning with „0") and remote version stamps, beginning with an identifier of a remote site (different than „0"), and further containing the identifier of the remote database version (i.e. its version stamp, as viewed from the remote site). It is illustrated in Fig. 7. First (Fig. 7a), the database version derivation tree before its distribution is presented. Two computer sites S1 and S2 are considered, each of them storing several database versions and respective versions of the multiversion object A. Next (Fig. 7b), the derivation tree is devided into two independent derivation trees, stored on sites S1 and S2, respectively. The root database version is added to the derivation tree of site S1, to represent „empty" local objects (it is not reasonable to share „empty" objects across the network). One may say, that this newly created „root" database version is a copy of a „root" database version taken from site S2. This is, however, only a logical copy, because *nil* versions it contains need not be copied between sites. The version stamps are renumerated, according to the structure of the local derivation trees. At the same time, a special track is kept between „shadow" local and remote database versions. This is achieved by changing version stamps held in association tables, as illustrated in Fig. 7c and 7d. For example, version a3 on site S1 is taken from remote site S2, from database version 0.1.1 (version stamp S1.0.1.1).

It is worth to notice, that the above idea may also be used to implement lazy replication technique: first reference to a remote version stamp in the association table forces getting a physical version from the remote site, and changing the version stamp in the association table to a local one. It is illustrated on Fig. 8.

4.2 Adaptation to Vertical and Mixed Distribution

Solutions for horizontal distribution proposed above may be adapted in a quite straightforward way to vertical distribution. Due to the fact that all versions of a multiversion object are stored on one site, association tables are also stored locally. To access remote multiversion objects, their association tables must be physically or logically copied to the local site. To do this, a similar technique to the one described in Section 4.1 may be used, which consists in getting shadow versions in a lazy way, and changing version stamps from „remote" to „local" ones. Mapping between multiversion objects identifiers and computer site identifiers must be, however, in this case extended. Indeed, as object stored on the same site are assumed to be semantically related (e.g. by composition), it is possible to adapt object clustering techniques [Cha93] to directly map object clusters to sites. As database versions are not stored as a whole on computer sites, distributing the database version tree, as it is shown in Section 4.1, is not possible. This tree has to be encoded and fully replicated in each computer site, and changes have to be propagated anytime a dbv-transaction is performed.



A	version stamps
a0=nil	0
a1	0.1
a2	0.2
a3	0.2.1

c)

A on S2	version stamps
a0=nil	0
a2	0.1
a3	0.1.1

A on S1	version stamps
a0=nil	0
a1	0.1
a3	**S2.0.1.1**

d)

Figure 7. Distributing database version derivation tree
a) centralized version derivation tree
b) distributed version derivation tree
c) centralized association table of object A
d) distributed association table of object A

A on S1	version stamps
a0=nil	0
a2	0.1
a3	0.1.1

A on S2	version stamps
a0=nil	0
a1	0.1
a3	**0.2**

Figure 8. Association tables of object A after first reading of the version a3 on site S2

As mentioned in Section 3.3, mixed distribution consists in allowing both vertical and horizontal distribution. As a counterpart to this gain in flexibility, implementing this approach requires to integrate both techniques of vertical distribution and techniques of horizontal distribution. Association tables may be centralized or distributed. This depends on the way of mixing vertical and horizontal approach. If computer sites are assigned much more objects than database versions, i.e. the corresponding rectangle is much more wide than high, association tables should be distributed. Indeed, in this case, versions of a given object are, on average, distributed on many sites. On the contrary, if sites are assigned few objects but in many versions, centralizing association tables and exchanging messages for object version read/write is better suited.

Mixed distribution raise the particular issue of « object version location transparency ». Actually, objects are distributed on different sites, depending on which version. In case of remote access to a given object version, the local site must hold a map in order to determine which remote site is responsible for this object version. The data structure used for object location evoked above is no more sufficient, since one has to locate no more objects but object versions. However, it may be extended in a very straightforward way, due to the restriction described in Section 3.3., so called fixed object clustering. Indeed, it is the sufficient to add, for each object clusters, the list of computer sites where object belonging to the cluster have some versions.

5. Conclusions

This paper presents three different ways of distributing a multiversion database managed according to the database version model (Section 2) on the sites of a computer network. Depending on application needs, the granule used for distribution is one of the following: a single database version (horizontal approach), a multiversion object (vertical approach), an object version (mixed approach). Each solution is described at the application level (Section 3), as well as at the system level (Section 4), where extensions to the basic data structures and algorithms which support the database version model are presented.

This latest point proves the usefulness of the database version model which can be distributed in a straightforward way, without loosing any mechanism, and without substantial implementation problems. It is a consequence of one major characteristic of the database version model, namely that version management is implemented below object management. On the contrary, in most other version models in which version management is built on the top of object manager, the object version distribution is more complicated, since it is dependent on object distribution. Moreover, as explicit links are usually used to associate consistent object versions, horizontal distribution is very difficult, since it requires to replicate every shared component object version together with those links, and then causes tiresome work to maintain consistency across copies after each update. Thus, only vertical distribution is possible. Also, as versions of complex objects are physically bound to their components, lazy replication would be more complicated to manage.

The main advantages of the proposed approach are: adequation, flexibility and efficiency. The approach is adequate to the specificity of cooperative work in a distributed environment. The way multiversion database is distributed reflects the way people are working together, preserving independent (stand-alone) work when needed, as well as concurrency when results have to be gathered.

The second advantage is flexibility: depending on the application domain one of three proposed distribution models may be applied. As the mixed distribution generalizes both horizontal and vertical approach, it is possible to cover a wide range of application domains. Moreover, because the database version model is orthogonal to object model (i.e. it does not require any specific object-oriented features, except object identification, to be implemented), our solution can be adapted to any data model.

Efficiency, the third advantage, results from ad hoc data structures and algorithms: network transmission and data replication are minimized by distributing data, so that remote accesses occur rarely and do not require immediate transfers of huge amounts of data.

The usefulness of the proposed distribution techniques are going to be verified very soon in the Multiversion Object Manager (MOM), which is a prototype of the database modeled and managed according to the database version approach. Its recent version works in a restricted client-server architecture. A client station supports user applications only, while all data structures are centrally stored on the server, which is of course to be extended by the ideas proposed in this paper.

Further extensions will be given to the presented solutions. They mainly concern the integration of a database schema (e.g. object-oriented schema) with the distribution policy. First, composition and inheritance relationships between classes will be used to deduced sets of semantically related object versions that should be located on the same site. The issue is similar to object clustering and distributing in conventional monoversion object oriented databases. However, it is more complex since, as it is shown in [CJK91], the database version approach could be extended to maintain consistency between versions of classes and versions of objects. Second, the database schema itself must be distributed, including methods and possible integrity constraints. Again, if the schema itself is versioned, monoversion solution cannot be adapted in a straightforward way.

References

[AN91] R. Ahmed and S.B. Navathe, *Version Management of Composite Objects in CAD Databases*, Proc. SIGMOD, pp. 218-227, Denver, Colorado, June 1991.

[Atw86] T. M. Atwood, *An object-oriented DBMS for design support applications*, Proc. COMPINT, pp. 299-307, Montréal, Canada, September 1986.

[BD92] Bell D. and Grimson J., *Distributed database systems*, Addison Wesley, Reading Mass, 1992.

[CLMG93] Chabridon S., Liao J.C., Ma Y., Gruenwald L., *Clustering techniques for Object-Oriented Database Systems*, Proc. 38th IEEE Computer Society International Conf, 1993, pp 232-242.

[CK86] Chou H., Kim W., *A Unifying Framework for Version Control in a CAD Environment*, Proc. 12th VLDB Conf, 1986.

[CJ90] Cellary W., Jomier G., *Consistency of Versions in Object-Oriented Databases*, Proc. 16th VLDB Conf., Brisbane, Australia, 1990.

[CM86] M .J. Carey and W. A. Muhanna, *The performance of multiversion concurrency control algorithms*, ACM Transactions on Computers and Systems 4(4), pp. 338-378, November 1986.

[EC94] Estublier J. and Casallas R., *The Adele Configuration Manager*; In: *Configuration Management* , ed. by W. Tichy, Wiley and son, in Software Trend Serie, 1994.

[GMH95] Garcia-Molina H. and Hsu M., *Distributed databases*; in: *Modern Database Systems, The Object Model, Interoperability and Beyond*, W. Kim ed., Addison Wesley publish., Chapt. 23 -- pp. 477-493, 1995.

[KC87] Katz R., Chang E., *Managing Change in a Computer Aided Design Database*, Proc. 13th VLDB Conf., 1987.

[KCB86] Katz R., Chang E., Bhateja R., *Version Modeling Concepts for CAD Databases*, Proc. ACM SIGMOD Conf., 1986.

[KRW92] M.H. Kay, P.J. Rivett, and T.J. Walters, *The Raleigh Activity Model : Integrating versions, concurrency, and access control*, Proc. BNCOD, 1992.

[KS92] W. Kõfer and H. Schoning, *Mapping a version model to a complex-object data model*, Proc. Data Engineering, Tempe (Arizona), 1992.

[OV91] Oszu M.T. and Valduriez P., *Principles of Distributed Database Systems*, Prentice Hall 1991.

[Sarda90] Sarda N., *Extensions to SQL for historical databases*, IEEE Transactions on Knowledge and DataEngineering, vol. 2, Nr. 2, pp. 220-230, June 1990.

[TCGJSS93] Tansel A. U., Clifford J., Gadia S., Jajodia S., Segev A. and Snodgrass R., *Temporal Databases: Theory, Design, and Implementation*, Benjamin / Cummings, Database Systems and Applications, 1993.

[TOC93] G.Talens, C.Oussalah, and M.F. Colinas, *Versions of simple and composite objects*, Proc. 19th VLDB, Dublin, 1993.