2-1991

Concurrency Control, Version Management and Transactions in Advanced Database Systems

Jonathan Walpole
Portland State University

MunTuck Yap

Citation Details

"Concurrency Control, Version Management and Transactions in Advanced Database Systems," Jonathan Walpole and Mun Tuck Yap, OGI Technical Report CS/E 91-009, February 1991.

CONCURRENCY CONTROL, VERSION MANAGEMENT AND TRANSACTIONS IN ADVANCED DATABASE SYSTEMS

Jonathan Walpole (Oregon Graduate Institute),
MunTuck Yap (Servio Corporation).

1. Introduction

This document constitutes the final deliverable for the research project titled "An Investigation of Selected Issues in Transaction Mechanism Design for Object Oriented Databases." The document describes our ideas for extending the traditional transaction concept for use in object oriented databases, and concentrates specifically on providing an underlying model to support the concurrency control and version management aspects of the problem. The ideas presented here are not however restricted to the domain of object oriented databases. They are more generally applicable to database systems that require flexibility in their versioning and concurrency control policies.

As database systems become more widely used in non-traditional applications, such as design environments (i.e., CAD and CAE) and office systems, there is a growing strain on current database technology. These new applications require support for cooperative work, long transactions, version management, and composite object manipulation. Current database technology does not provide adequate support for any of the above activities. To make matters worse, the flavor of support and the precise policies required may vary between different applications or within the same application. A single transaction mechanism, which defines one policy for concurrency control, recovery and version management is unlikely to support a sufficiently wide range of applications. We believe that a better approach is to provide (a) a set of general mechanisms for concurrency control, version management, and recovery, and (b) a facility that allows applications to define their own policies. The database may, of course, provide a set of predefined policies for those applications that do not wish to implement special policies of their own.

We also believe that the underlying mechanisms provided for concurrency control, version management and recovery should be integrated. The problem that we are addressing in designing these mechanisms can be more clearly defined as a problem of providing support for the control, propagation, and representation of change in a database. In current database systems this problem is addressed, at least in part, by the use of an atomic transaction mechanism and a higher level version management layer. However, in such systems the lack of integration between transactions and version management results in ineffective and inappropriate concurrency control because the transaction mechanism can only detect conflicts on current (not pending) versions. In a system which supports multiple versions of an object, each of which may be immutable, it makes little sense to use concurrency control mechanisms in order to prevent modification of existing versions. It is more useful to be able to apply concurrency control on the version history as a whole (or parts of it) in order to prevent the creation of multiple conflicting new versions.

In addition, the classical transaction model does not provide adequate support for long transactions because it forces concurrency control and recovery to be applied at the same granularity, and

it imposes a single fixed policy for concurrency control which encourages competition for resources rather than cooperation.

In this document we define a model for change control, propagation and representation. A key aspect of the model is the separation of mechanism from policy. We describe an integrated collection of basic mechanisms that underlie version management, concurrency control, and change propagation. These mechanisms allow a variety of concurrency control, version naming, and change propagation policies to be implemented, and even support multiple different polices within the same system. The salient features of the model are:

i. Support for both immutable and mutable objects.
ii. Support for application-defined policies for concurrency control, version creation, version naming, and change propagation.
iii. The use of database regions to support various forms of sharing within groups.
iv. The integration of concurrency control with the region concept.
v. Support for long and cooperative transactions.

The following two sections describe the model and outline its support for a variety of different versioning and concurrency control policies. In section 4 we briefly discuss the effect of our model on various aspects of schema evolution, and in section 5 we outline future work. Finally, section 6 concludes the report.

2. The Version Model

The following sub-sections define a number of basic and generic mechanisms for version management. These mechanisms are intended to support a variety of different versioning policies.

2.1. Mechanisms for Supporting Versions

All objects in the model are, by default, versionable. However, objects with only a single version will be common (therefore, we provide special support for this case). Every version has a *version history* consisting of a list of versions that represent its evolution. For the initial version of an object this list is empty. The version histories for all versions of the same conceptual object (i.e. the set of versions representing the evolution of a single entity) are combined and stored as a directed acyclic graph (DAG). The DAG for each conceptual object is stored in an associated *history object.* The use of distinct history objects, as opposed to storing version history information with the versions themselves, has the advantage that it allows information associated with the version histories to be accessed without having to retrieve the individual versions. We make extensive use of this feature in the concurrency control and versioning mechanisms described in later sections.

2.2. Referring to Versions

Every version (and history object) is considered to be a distinct object and has a unique object identifier. An object's state is composed of information stored directly in the object (called its *internal state*), and references to other objects.

The model defines two mechanisms for referring to objects. The first, and simplest, approach

is to use the object's identifier directly. This type of reference is called a *direct reference*. The second approach is to refer to a version of an object indirectly via its history object. This type of reference is called an *evaluated reference* because it requires an operation on a history object in order to determine the object identifier for the requested version (evaluated references are only applicable to versioned objects). For example, evaluated references can be used to access "the latest released version", or "the most recent version on a particular path," without requiring the correct object identifier to be known in advance.

History objects support a wide variety of version naming policies by allowing *tags* to be associated with object identifiers in their version history graphs. Evaluated references consist of the object identifier of a history object, and a tag that is associated with a version within that history object. In order to dereference an evaluated reference, a history object searches its version history graph for the specified tag and returns the associated object identifier for the tagged version. History objects also allow tags to be moved between versions in the version history. In this way, the most recent version on a particular path can be tagged by moving the appropriate tag every time a new version, derived from the tagged version, is added to the version history. Note, the history object does not associate any semantics with tags. The responsibility for moving a tag representing the most recent version on a particular path rests with higher level software.

Figure 1: The structure of an evaluated reference

In summary, history objects support the following tag-related operations:

new_tag = tag (object_id_of_version)
status = move_tag (tag, object_id_of_version)
object_id_of_version = evaluate (tag)

2.3. Version States

Versions exist in one of three possible states: *working*, *static*, and *frozen*. Working versions are mutable (i.e. they can be updated in place), whereas static and frozen versions are immutable (i.e. they can only be accessed for reading). When an object is created, its initial version is a working version. The internal state and references within a working version can be updated in place. Working versions can be converted to static versions using the f*ix* operation. Fix causes the version to be marked read-only. That is, neither its internal state nor its references can be updated. Note, however, that static versions can contain evaluated references. A static version can be converted to a frozen version using the freeze operation. Freeze causes the version's evaluated references to be converted to direct references.

Fix and freeze can be combined to support a wide range of strategies for freezing the state of composite objects. For example, one extreme would be to apply fix and freeze to all the objects

reachable from a particular object. This would implement a “deep-freeze” of the object (also referred to as the creation of a *version slice*).

New working versions can be created (copied) from other versions using the *derive* operation. Derive creates a new working version by copying the specified version, and then updates the associated history object in order to add the new version to the version history as a child of the original version. The various version states and the operations that can be applied to them are illustrated in figure 1.

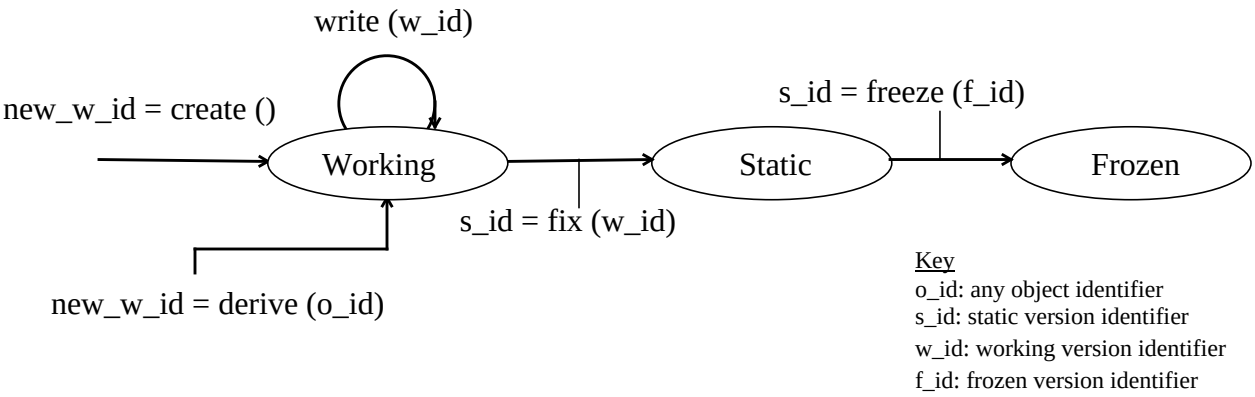

Figure 2: State transition diagram for versions

2.4. Change Propagation

Change propagation is concerned with the semantics of inter-object references and, in particular, the effect of modifications to referenced objects on referencing objects. The term “modifications” is used here to represent the creation of new versions as well as direct updates (in place) to objects. Similarly, the “effect” of a modification could be that the modification is visible by following the reference, it could cause the creation of a new version of the referencing object, or it could result in a direct update to the referencing object.

Many different semantics can be associated with inter-object references. One particular type of reference that is often distinguished is the *part-of* relationship between components of *composite objects*. This distinction is often used to influence policy decisions regarding concurrency control, versioning and change propagation. For example, changes may be propagated across *part-of* references but not across other references, or concurrency control may be automatically applied to all the objects that are part of a higher level composite object. The model described here does not make such semantic distinctions because we believe that they are too simplistic and that many applications require reference semantics that are considerably more complex than can be captured at this level of abstraction. Instead, our version operations and reference types provide a flexible set of alternatives for change propagation (the integration of concurrency control with these alternatives is discussed in later sections).

Composite objects can implement their own policies for change propagation by making an ap-

propriate choice of reference type (either direct or evaluated), and by selectively applying the fix and freeze operations for each of their components. For example, if a composite object does not want changes to be propagated it should contain only direct references to frozen versions. If changes are to be propagated, the object can contain direct (or evaluated) references to working versions. A more controlled form of change propagation can be supported by using evaluated references to fixed versions. In this latter case, changes are only propagated when tags are moved. Note that these choices are available, independently, for every reference and component object within a composite object.

The model does not currently support the propagation of version creation (sometimes called percolation). In other words, to allow the creation of a new version of a referenced object to cause the creation of a new version of the referring object. We are investigating extensions to the evaluated reference mechanism to support this facility.

3. Concurrency Control

Conventional concurrency control mechanisms, based on the concept of serializability, are of limited use in many database applications. In this section, we present a more flexible model of concurrency control in which a collection of basic mechanisms can be used to implement a variety of different concurrency control policies.

The proposed concurrency control model divides the database into a number of distinct *regions*, provides mechanisms for entering and removing objects in/from regions, and supports region-specific policies for concurrency control.

3.1. Database Regions

The database is divided into a number of distinct regions. There are three different types of region: a single *global* region; multiple *group* regions; and multiple *private* regions. The single global region is shared by all users of the database and contains objects that are visible to everyone. The global region is created when the database is initialized. Group regions are created dynamically and contain objects that are visible to a limited number (the group) of users. Throughout the course of a project or application we would expect many of these group regions to be created to support temporary cooperation and later to be deleted. Private regions are associated with individual users, and are created on a per session basis (sessions may span login boundaries).

3.2. Using Regions

All object creation and manipulation takes place in private regions, i.e. if a user wants to create an object, the object is created within that user's private region. In order to enter the newly created object into either a group region or the global region the user must perform a *checkin* operation. In other words, checkin is the mechanism by which objects are entered into the global or group regions. The checkin operation has *move* semantics. That is, it causes the object to be removed from the source region and entered into the target region. The object maintains its object identity.

In order to access an existing object, a user must first enter the object in his/her private region. This applies to all operations (reads, writes, fixes, and freezes). The mechanism by which objects are entered into private regions is called *checkout*. For fixed and frozen versions checkout has *copy on write* semantics -- by this we mean that a new version is *derived* and entered into the region only if the object is checked out for write access. Note that fix and freeze are treated as read operations in this respect.

3.3. Concurrency Control Using Checkout/Checkin

A selection of basic concurrency control mechanisms are built into the checkout and checkin operations. These mechanisms are automatically invoked when objects are checked in and out. When an object (working, fixed, or frozen) is checked out for reading, a read lock is placed on that object. Read locks have the following semantics: they prevent write locks from being acquired, but allow further read locks to be acquired. When an object is checked out for writing, the behavior of checkout depends on the type of the object. For working versions, a write lock is applied to the checked out version. Write locks prevent read locks or other write locks from being acquired. For fixed and frozen versions, a new version is derived and a write lock is applied to this version leaving the original version untouched.

In addition to these basic mechanisms, further concurrency control mechanisms can be selectively invoked on checkin and checkout. When used in conjunction with the database region model (described above) these concurrency control mechanisms are flexible enough to support extended transaction models for long and cooperative transactions, and can implement a wide range of concurrency control policies.

The expected mode of operation for the system is as follows:

i. A user performs checkout operations on one or more objects. During each checkout operation a concurrency control mechanism is selectively invoked (details of the concurrency control mechanisms are presented later in this section).

ii. The user then manipulates the objects in his/her private region in isolation from the rest of the database. Eventually, some objects are ready to be checked in to the global (or group) region. These will appear as new versions or as changes to working versions.

iii. The checkin operation for each object involves a concurrency control check. If there are no concurrency control conflicts the objects are checked in and the concurrency control mechanisms that were activated by that user (during an earlier checkout) can be deactivated.

The goal of concurrency control is to provide applications with control over changes to shared

data. In our versioning model, only data in group or global regions can be shared, and there are only two ways in which an application can share data with another application:

i. it can hold a direct reference to an object that is referenced by another application.

ii. it can hold an evaluated reference to an object that is referenced by another application.

Changes to shared data are observable only in the following cases: (a) when applications share a reference (direct or evaluated) to a working version; and (b) when applications share an evaluated reference and the associated tag is moved from one version to another. Consequently, applications wishing to observe changes to shared data must either keep that data in a working version or, if the data is in fixed or frozen versions, they must share the same tag and move the tag to reflect changes that occur through the creation of new versions.

In light of the above, the model defines the following concurrency control mechanisms:

Tag locks: history objects allow users to associate a lock with a named tag. When set, tag locks have the following effect (a) they prevent anyone (expect to holder of the lock) from moving the tag to a new version, (b) they return a warning when anyone except the holder of the lock attempts to checkout the tagged version, and (c) they return a warning when an attempt is made to add a new version as an immediate child of the tagged version.

Change detection: optimistic concurrency control policies require the ability to detect changes that have occurred to the objects in their read and write sets. In the proposed model, change can only occur in two ways: through the movement of a tag, and through the modification of a working version. In order to detect such changes, state information must be recorded at checkout time and, subsequently, examined at checkin time. During the checkout of a working version, the value of a counter associated with the version is recorded. A subsequent checkin of the same working version causes the recorded counter value to be compared with the current counter value. If the values are not equal a warning is returned. Similarly, for tagged frozen versions the position of the tag is recorded at checkout time and, subsequently, compared at checkin time. If the tag has been moved a warning is returned. The warnings are interpreted by higher level software that implements the concurrency control policy.

3.4. Policies for Concurrency Control

The application-level interface to the proposed system is encapsulated in one or more policy-enforcing entities called *agents*. Agents invoke checkout and checkin operations on behalf of applications, specify the concurrency control mechanisms to be invoked, and define the policies to be used for handling the various mechanisms described above. Each agent encapsulates a specific concurrency control policy, or transaction style, and defines an appropriate interface to applications. Several agents can coexist in the system at the same time, and we envisage a typical system providing a generic set of agents that implement a selection of useful concurrency control policies.

This flexibility is important for supporting many advanced database applications.

Tag locks, and the change detection mechanism can be used to support a variety of concurrency control policies. Some possible uses of the mechanisms are described below:

Using tag locks: when set, tag locks prevent anyone (expect to holder of the lock) from moving the tag to a new version. However, they do not prevent a new version from being checked in as a child of the tag-locked version. A warning is returned in this case, however, agents can choose to ignore it. If necessary, the warning can be used to trigger some other action, such as a mail message, or a notification that a merge may be required in the future. Similarly, the warning that is returned when anyone except the holder of the lock attempts to checkout the tagged version can be ignored, or can be used to trigger other actions.

Using change detection: agents implementing optimistic concurrency control policies can use the change detection mechanisms to detect conflicts at checkin time. The flexibility provided by this mechanism arises from the fact that the detection of a conflict does not necessarily prevent data from being checked in. In the case of tag conflicts, agents are free to checkin their objects as new versions. These new versions are no longer on the tagged path within the version history. However, the warning returned as a result of the conflict can be used to trigger future, compensating actions.

3.5. Extended Transaction Concepts

From a concurrency control perspective, the model's support for multiple versions, flexible concurrency control policies, and database regions, makes it well suited for supporting long and cooperative transactions. In fact, we suggest that many new database applications (such as CAD, software engineering, office systems) have a strong requirement for transactions that are both long and cooperative.

Long transactions can be supported by allowing users to work on new versions in their private regions whilst allowing access to older versions in the group or global regions. Since the concurrency control mechanisms are built with knowledge of the version and database region model, they can be used to implement a variety of long transaction concurrency control policies, such as: triggering e-mail to/from the person who is using the object in a long transaction; warning when future merges may be required; and allowing (or disallowing) alternate versions to be checked in.

The group region facility, combined with the flexible concurrency control mechanisms that are associated with its checkin and checkout operations, provides useful support for cooperative transactions. The partial results of one transaction can be selectively released from a user's private region to the other members of a group by checking versions into the group region. Only the final results of the cooperative transaction become globally visible when they are checked in to the global region. Furthermore, users involved in long cooperative transactions can use one set of concurrency control mechanisms to checkin/checkout objects to/from the global region, and a separate set of mechanisms for checkout and checkin between their private regions and the group region. This allows a long cooperative transaction to support intra-group cooperation at a more integrated level than the interaction between the group members and the other database users.

The model presented here does not address recovery problems. The recovery aspects of long and cooperative transactions are a subject for future research.

4. Versioning and Schema Evolution

Schema evolution is one of the thorny problems in object oriented database research. Some aspects of the schema evolution problem are related to versioning models. Therefore, in this section we outline the relationship between our versioning model and the schema evolution problem.

First, it is important to distinguish between the action of versioning instances and versioning classes. The former is not a schema evolution issue whereas the latter is. Our versioning model defines mechanisms that can be applied uniformly to all objects (either instances or classes). Second, it is important to distinguish between creating a new version of a class and creating a subclass. In both cases a new class is created. However, the DAG representing the version history of the class does not support any notion of inheritance. Instead, it is merely a mechanism for representing the evolution of a conceptual object.

Therefore, the model enhances conventional schema evolution facilities by allowing classes to be versioned. The issue of whether or not to allow a particular change to a class (through the creation of a new version) is not the concern of the basic mechanisms for versioning. This should be handled by higher level software that has more semantic knowledge. Similarly, the system may provide mechanisms, at a higher level, for migrating instances between the old and new versions of classes. The questions of when to perform such a migration, and how to convert instance structures so that they are compatible with the new class are policy issues which should not affect the versioning model we have proposed. However, we must ensure that the basic mechanisms provided by the model do not prevent such migrations. For example, the relationship between an instance and its class could be represented using a direct or an evaluated reference. The use of a direct reference is the simplest case, but requires a higher level mechanism to migrate instances to new classes. Evaluated references, however, provide the basis for an automatic migration mechanism. For example, if an instance references its class using an evaluated reference, and if a new version of the class is created and tagged appropriately, subsequent references from the instance to its class will be directed to the new class. Such a mechanisms could be used to automatically migrate instances to new versions of classes as long as structural changes to the instance are not required.

5. Future Work

Several important questions (listed below) remain unanswered at this stage of our research. The following list defines topics for future work:

Atomic Operations:

- The model must be extended to support atomic operations on history objects, and atomic checkins and checkouts of multiple objects.

Automatic Checkout and Checkin:

- How can checkout and checkin be modified so that they propagate across references?
- Can objects be checked in without releasing concurrency control mechanisms (i.e. locks), i.e., is the release of locks an orthogonal issue to checkin?
- The interface to the checkout and checkin operations must be more precisely defined.
- A more precise definition of the location and storage of information used during checkout and checkin operations is required, i.e., counters, private region tables, the structure of history objects.

Percolation:

- An important extension is needed to allow automatic version creation to be propagated across references (lets call this percolation).

Delete:

- Do we need a delete operation? If so, what semantics does it have, and how is it propagated across references?

History Objects:

- We need a more rigid definition of the structure of a history object. In particular, we need to define a structure that enables it to be searched efficiently for a given tag.

Policy agents:

- We need a precise definition of the interface that can be used by policy agents. It is also not clear how much of the system internals need to be exposed to agents in order for them to be efficient.
- How do different concurrency control policies interact? i.e. which agents can reasonably coexist simultaneously?

Recovery:

- A key issue in extending the classical transaction concept is the separation of issues of recovery from issues of concurrency control. The classical transaction concept binds both issues together in the single concept of atomicity, requiring both concurrency atomicity and failure atomicity to be provided at the same granularity. The model described here should be extended to incorporate recovery in a manner that is orthogonal to the concurrency control aspects of the model. In particular, during long transactions users should be able to commit results from within private regions without making those results visible to the outside world.

Distribution:

- What are the implications of the model for operation in a distributed environment?

6. Conclusion

We have proposed a model that provides flexible and integrated support for version management and concurrency control in advanced database systems. The model allows versioning policies, concurrency control policies, and change propagation to be specified according to application requirements. Versioned and non-versioned objects are supported, and mutable (working) versions can exist at the leaves of a derivation history. The concept of evaluated references provides flexible control over data sharing, and the proposed mechanisms for version management, database regions, and concurrency control are powerful and intuitive. The combination of these mechanisms enables long and cooperative transactions to be supported. We believe that the flexibility offered by our model is necessary to meet the demands of future (and current) database applications.