Recovery in Multiversion Objectbase Systems

Blanca H. Perez
Department of Computer Science
California State University, San Marcos
San Marcos, CA, U. S. A

Dr. Ahmad Reza Hadaegh
Department of Computer Science
California State University San Marcos
San Marcos, CA, U. S. A

Abstract

Objectbases are becoming more popular because they reflect the real world more accurately and realistically than Relational Databases. Multiversioning of object-oriented systems uses previous versions of the objects in order to enhance the performance of the transaction management. An optimistic algorithm to manage concurrent execution of the transactions in a centralized multiversion objectbase environment is presented by Hadaegh and Barker [3]. In that model each transaction obtains a copy (version) of all objects it needs to execute. If it is decided that a transaction can be committed, the associated versions become persistent objects and the transaction commits. We propose a recovery algorithm for the Hadaegh and Barker model.

Keywords: Recovery, Concurrency Control, Objectbase Systems, Nested Transaction

1. Introduction

Two main aspects of the transaction management are concurrency control and recovery. Concurrency control ensures that individual users see consistent states of the database even though operations on behalf of many users may be interleaved by the database management system; i.e., to the external world, the database appears as if the transactions were executed one at a time in some order. Recovery ensures that the database is fault tolerant; i.e., the data are not corrupted even in failure. A concurrency control algorithm guarantees serialization of a set of concurrently executing transactions based on some correctness criterion. A set of transactions is serializable if their execution history is equivalent to a serial execution of those transactions. The most common correctness criterion is conflict serializability [1]. This paper uses the concept of value-serializability created by Hadaegh and Barker [3].

Concurrency control is divided into two broad categories: optimistic and pessimistic. Pessimistic protocols block the transactions by deferring the executions of some conflicting operations. Optimistic algorithms do not block the transactions but validate their correctness at commit time.

A reliable DBSM must be equipped with a recovery algorithm. The recovery algorithm must ensure that the database is brought into a consistent state after a failure occurs. Failures can be divided into three categories: *Transaction Failure, System Failure,* and *Media Failure.* A transaction failure is detected by the application, e.g., encountering division by zero. A system failure (crash) occurs when the system fails in a way that causes the loss of volatile memory contents. An example of system failure is power failure. Media failure occurs from the breakdown of persistent storage and potentially causes the loss of data on non-volatile storage. If the failure occurs, the recovery algorithm is executed to bring back the system into a consistent state.

Recently research has moved toward object-oriented databases with nested transactions. An objectbase consists of a set of objects that contain structure and behavior. The structure is the set of attributes encapsulated by the object. An object's behavior is defined by a set of procedures called methods. A method's operations can read or write an attribute, or invoke another method.

In the nested transaction model [7], a transaction contains any number of subtransactions, and each subtransaction again may compromise any number of subtransactions. This paper adapts the nested transaction model used in Hadaegh and Barker's model [3]. They introduce two types of transactions: user transactions and version transactions. A user transaction is a sequence of method invocations on a set of objects. A version transaction contains read/write operations on a version of an object and any nested method invocations.

Hadaegh and Barker developed the concurrency control algorithm in their

multiversion objectbase system. The goal of our research is to enhance their model with a recovery algorithm.

This paper makes the following contributions:

- It enhances the Hadaegh and Barker's architecture. We add the Recovery Manager components to the architecture to recover the system in case a transaction or a system failure occurs.
- We propose a recovery algorithm that reflects the expanded and enhanced architecture. The algorithm shows how the version objects are used to minimize the overhead during normal transaction processing and to minimize the time required to do the recovery.

This paper is organized as follows: In Section 2, we describe the related work. In section 3, we briefly explain the version model, and the transaction model. Then we show the extended architecture that incorporates the Recovery Manager in Section 4. In Section 5, we introduce the recovery algorithm. Finally, we conclude this paper with some closing thoughts and address related open problems.

2. Related Work

In this section, we briefly review relevant multiversion, objectbase concurrency control, and recovery literature. Multiversioning allows for enhanced concurrency, simplifies recoverability, and supports temporal data management. Some of the related work that directly lead to our work are Graham and Barker [2], Hadaegh and Barker [3,4,5], and Wieler and Barker [8].

Graham and Barker [2] proposed an optimistic concurrency control scheme for objectbase systems. In their algorithm, each transaction obtains copies of the objects it requires and is executed independently of other transactions.

Hadaegh and Barker [3,4,5] illustrated an optimistic concurrency control for multiversion objectbase systems. They used closed nested transactions where updates of uncommitted transactions are never present in the objectbase at commit time. They presented a serializability theory called *value-serializability* and an architecture that could be used as the basics for the development of optimistic concurrency control protocols on a centralized objectbase environment.

Wieler and Barker created a method and object scheduler algorithms that satisfy the operation ordering criteria of reliability definitions. To support the reliable schedulers, a novel logging scheme is introduced where each object maintains its own logical log. An object's logical log contains log records for each action of the object transactions executing on the object. An object's updates log records, grouped together in a logical log, promote efficient recovery from transaction aborts and system failures.

Nakajima [6] introduces a recovery protocol with multiversion objects using branching multiversion objects instead of undo operations. Nakajima introduces an alternative structure for the versions of the objects called branched multiversion object. The execution of branched multiversion objects is structured in the trees of versions and may create several versions during one execution of methods. For recovery, specifically, for transaction aborts, compensating methods are proposed to undo the effects of their corresponding methods. The recovery algorithm search the version created by an aborted action, and select successor of this versions. If this version is the current version, discard the version and its successors, and the version before the current version becomes the current version.

3. The Model

This section defines objects, versions, and transactions.

3.1 Object and Version Model

In Hadaegh and Barker's model [3], an objectbase consists of a set of objects. An object has a set of attributes that determine the state of the object and a set of methods that can change the behavior of the object. One or more versions (copies) of a particular object may exist at a particular time.

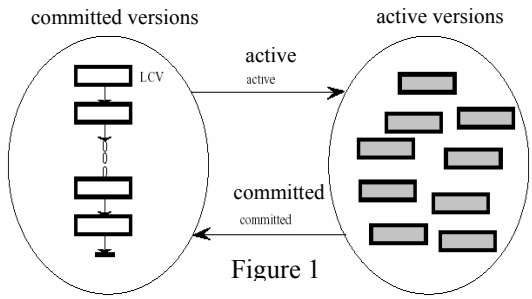

Figure 1

The versions of the objects are either committed or active. Figure 1 shows the structure of the committed versions and the active versions. Committed versions are maintained in a data link structure called the *version-chain* of the object. They are in chronological order depending on some correctness criteria. The most recent committed version of the object is on the head of the chain and is called the *last committed version* (LCV). An active version of the object is made (copied) from the latest committed version of the object and is manipulated independently of all other versions. The active versions can be committed after the correctness specification for concurrency control is applied to determine if and where in the version-chain the active version can be inserted.

An object is an ordered triple, $o=<f,A,M>$, where f is a unique object identifier; A is the object's structure, composed of attributes; and M is the object's behavior, composed of methods. An object with identifier f is denoted o^f. Objects are versionable in that several versions can be derived from a given object. Versions are either active or committed. An active version i of an object o^f (denoted v^{fi}) begins as a copy of the object which can then be manipulated independently of all other such versions. The modified active version is promoted to a committed version if its state is consistent with other committed versions in object family f; otherwise, the active version is modified again, and if it still cannot be promoted to a committed object, it is disposed. A new committed version is inserted in an appropriate position in the version-chain, which is specified by the correctness criteria. Periodically, the older committed versions are removed from the version-chain and archived due to the limitation of the size of the version-chain. The version-chain of an object effectively captures the evolution of the objects (historical information through time).

3.2 Transaction Model

The transactions are submitted to the objectbase by multiple, concurrent users. A transaction submitted by a user consists of a set of method invocations. Methods can invoke other methods. The nested methods invoked by the users are executed concurrently. The nested transactions submitted by the users may be divided into two groups. The first group includes top-level transactions explicitly created by the users. They are called *user transactions*. The second group contains transactions generated from the method invocations made by the top-level transactions. They are called *version transactions*.

The operations of a user transaction are method invocations that are transformed by the system into version transactions. Version transactions manipulate the active versions of the relevant object.

4. The Architecture

This section explains the architecture developed by Hadaegh and Barker [3]. It also illustrates our recovery management extension that brings reliability to the system.

Figure 2 shows three original main components of the architecture: the *Transaction Processor*, the *Version Processor*, and the *Validation Processor*. The Transaction Processor contains two components: the User Transaction Manager and the Method Scheduler. The User Transaction Manager coordinates the execution of user transactions (UT_i) by converting the method invocations of UT_i to version transactions and passes them to the Method Scheduler. The Method Scheduler permits concurrent execution of a version transaction of UT_i based on the information obtained from static analysis [2] so that version transactions of UT_i invoked on the same active version are ordered, enforcing intra-UT concurrency control. Next, the Method Scheduler sends the version transactions and their schedule to the Version Processor.

The Version Processor also consists of two components: the Version Transaction Manager and the Execution Manager. For each version transaction of the user transaction UT_i executing on object f, the Version Transaction Manager obtains a copy of the last committed version of the object family f in the objectbase and puts it in the Unstable Store. This copy is the active version v^{fi}. Next, it passes the versions transactions to the Execution Manager. The Version Transaction Manager builds a version list for all versions of objects associated with UT_i; this list is denoted by $VRLST_1(UT_i)$. It also builds a version list for the last committed version of the objects associated with UT_i; denoted by $VRLST_2(UT_i)$. When all version transactions of UT_i are completed, the Version Transaction Manager sends $VRLST_1(UT_i)$ to the Execution Manager. The Execution Manager executes the operations of the version

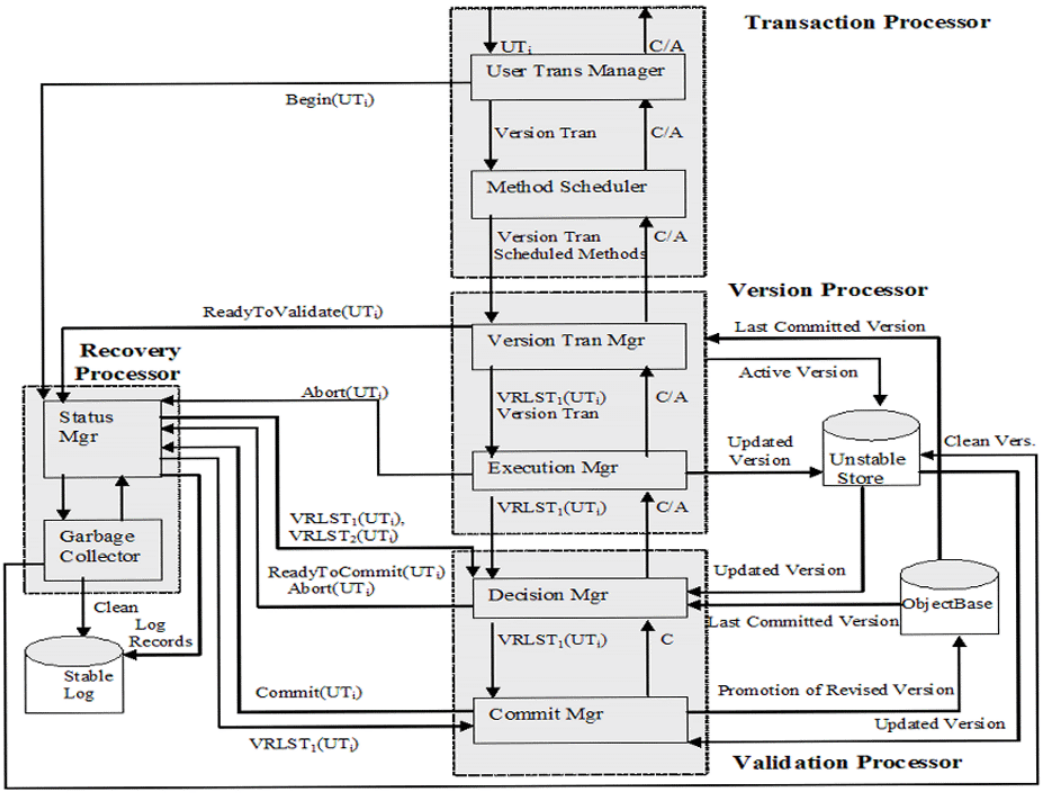

Figure 2

transactions against the active versions in the Unstable Store. When the Execution Manager obtains $VRLST_1(UT_i)$ from the Version Transaction Manager, it passes $VRLST_1(UT_i)$ to the Validation Processor.

The two main components of the Validation Processor are the Decision Manager and the Commit Manager. For each f in the $VRLST_1(UT_i)$, the Decision Manager compares the updated active version v^{fi} with the last committed version of object family f (o^{fn}), and determines if an updated active version would create inconsistency in the objectbase. The state of an updated active version v^{fi} is consistent with the states of committed versions in the object family f if the values of the attributes read by UT_i in v^{fi} have not been modified in the objectbase during the lifetime of UT_i. If the states of all active versions are consistent with the states of their corresponding committed versions in the objectbase, $VRLST_1(UT_i)$ is sent to the Commit Manager; otherwise, some active versions are invalid and UT_i should be aborted or reconciled.

If it is decided that UT_i is to be committed, the Commit Manager promotes the revised versions to committed versions and merges the committed versions with their corresponding objects in the objectbase, thereby creating new states for the objects in objectbase. Next, the Commit Manager sends a commit message to the Decision Manager. The commit message is eventually passed to the user by the User Transaction Manager.

As shown in the architecture, we have added a new component called the Recovery Processor to take care of the recovery issues in case the system fails. The two components of the Recovery Processor are the Status Manager and the Garbage Collector. During the execution of UT_i, the other three processors, explained above, report the status of the transaction to the Status Manager. Every time the Status Manager receives a new status for UT_i, it records it in the log. The Status Manager keeps separate records for each transaction in the log. Each record of the log contains four fields.

UT_i	$VRLST_1(UT_i)$	$VRLST_2(UT_i)$	*Status of* UT_i

The first field is reserved for identification of the transaction. The second field is used to store the identification of the active versions that are being processed by the transaction. The third field is used to record the identification of the base (original) versions of the active versions processed by the transaction. Finally the last field refers to the current status of the transaction.

The Garbage Collector cleans up the log and the Unstable Store by removing any information recorded for committed or aborted transactions. The next section explains the Recovery Processor in detail.

5. The Recovery Management

5.1 Main Operation

The Status Manager contains five major methods: *Begin*(UT_i), *ReadyToValidate*(UT_i), *ReadyToCommit*(UT_i), *Abort*(UT_i), and *Commit*(UT_i).

Begin(UT_i)*:* This marks the beginning of the user transaction UT_i. The User Transaction Manager invokes this method. When the Status Manager obtains the *Begin* message, it creates a record in the log for UT_i with *Begin* status.

UT_i	?	?	*Begin*

ReadyToValidate(UT_i)*:* This marks the pre-commit of all version transactions of the user transaction UT_i. The Version Transaction Manager invokes this method after all version transactions of user transaction UT_i are pre-committed. In this method, the Version Transaction Manager passes the $VRLST_1$ (UT_i), and the $VRLST_2$ (UT_i) to the Status Manager. When the Status Manager receives the *ReadyToValidate* message, it adds the two version lists to the UT_i record in the log and changes the status of UT_i to *ReadyToValidate*.

UT_i	$VRLST_1(UT_i)$	$VRLST_2(UT_i)$	*ReadyToValidate*

ReadyToCommit(UT_i)*:* This denotes that the user transaction UT_i has successfully passed the validation status and is ready to commit. The Decision Manager invokes this method after it determines that the updated active versions will not create inconsistency in the objectbase if they become persistent objects. When the Status Manager receives the *ReadyToCommit* message, it changes the status of the transaction to *ReadyToCommit* in the log record for UT_i.

UT_i	$VRLST_1(UT_i)$	$VRLST_2(UT_i)$	*ReadyToCommit*

Abort(UT_i): This denotes the unsuccessful completion of the user transaction UT_i. UT_i aborts if either it terminates abnormally during the execution (transaction failure) or it fails to pass the Validation test done by the Decision Manager. In the former case, the Execution Manager sends the *Abort* message to the Status Manager. In the latter case, the Decision Manager notifies the Status Manager about the abortion of UT_i. When the Status Manager receives an *Abort* message, it changes the status of the transaction to *Abort* in the log for UT_i.

UT_i	$VRLST_1(UT_i)$	$VRLST_2(UT_i)$	*Abort*

Commit(UT_i)*:* This marks the successful completion of the user transaction UT_i. The Commit Manager calls this method after all active versions have been successfully promoted to the top of the version-chain of the objects that have been used by UT_i. The Commit Manager sends a *Commit* message to the Status Manager. When the Status Manager receives the *Commit* message, it changes the status of the transaction to *Commit* in the log record of UT_i.

UT_i	$VRLST_1(UT_i)$	$VRLST_2(UT_i)$	*Commit*

During normal transaction processing, the Status Manager only records in the log the status of the transaction and the version list. All operations make by the transactions are done against the active versions. So the versions of the objects minimize the overhead of the log during normal transaction processing.

5.2 Recovery Algorithm

If a system crash occurs, the recovery is done as follows: The Status Manager performs an analysis on the log, line by line to find out the status of the transactions. If the status of the transaction is *Begin*, it changes the status to *Abort,* because the transaction was not completed before the crash occurred. If the status of the transaction is *ReadyToValidate*, it restores the $VRLST_1$ (UT_i) and $VRLST_2$ (UT_i) from the transaction record and sends these lists to the Decision Manager. If the status of the transaction is *ReadyToCommit*, it restores the $VRLST_1$ (UT_i) from the transaction record and sends this list to

the Commit Manager. Any record with *Commit* or *Abort* status remains unchanged.

The recovery algorithm minimizes the time required to do the recovery because of the lacking of undoing or redoing processing and the presence of the active version in Unstable Store.

This algorithm differs from traditional multi-version recovery algorithm in that the recovery is accomplished base on the current status of the transaction recorded in the log.

5.3 System Clean-up

Periodically, the Status Manager invokes the Garbage Collector to clean up the log. The Garbage Collector sequentially goes through the log and removes the records that refer to aborted or committed transactions. In addition, for every record of a transaction such as UT_i that is removed from the log, the Garbage Collector removes all the active versions associated with UT_i from the Unstable Store. The Status Manager invokes the Garbage Collector during the idle periods, because clean-up of the Unstable Store and the log can be a particularly time-consuming task that slows down the system.

The recovery algorithm is idempotent. If the system crashes again during restart, the recovery algorithm performs exactly the same steps as it did during the previous restart. Further, our Recovery Processor enables fast crash recovery while incurring low overhead during normal transaction processing. In the recovery algorithm, the recovery time is very fast since the active versions and the version-chain are available in the Unstable Store and objectbase respectively. This method avoids writing undo and redo log records. This significantly reduces the storage overhead for the log compared with other log-based schemes normally used in traditional databases.

6. Conclusions and Future Work

This paper presented the recovery algorithm based on the assumption that the transaction aborts in case of conflict. One area for immediate future works is implementing recovery using simple and complex reconciliation to re-execute the transaction instead of abort it. We can also extend this work to investigate concurrency control and recovery using both pessimistic and optimistic techniques in distributed objectbase systems.

7. References

[1] Bernstein, P.A., Hadzilacos, V., Goodman, N. "*Concurrency Control and Recovery in Database Systems"*, Addison-Wesley Pub. Co., 1987.

[2] Graham, P., and Baker, K. "*Effective Optimistic Concurrency Control in Multiversion Object Bases"*. In Proceedings of International Symposium on Object Oriented Methodologies and Systems (ISOOMS), volume 858, pp 313-328. In Springer-Verlag. Lecture Notes in Computer Science, September 1994.

[3] Hadaegh, A. and Barker, K. "*Version Manager in Object Based System, Advances in Data Base and Information System"*, pp 126-133, Moscow, September 1996.

[4] Hadaegh, A. and Barker, K. "*Simple Reconciliation of Transactions to Increase Concurrency in Historical Object-base"*, XII Brazilian Symposium on Database Systems – SBBD'97, pp 44-64, October 1997.

[5] Hadaegh, A. and Barker, K. Partial Re-execution. "*Reconciling Transactions to increase Concurrency in Object-bases",* International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Monte Carlos Resort, Las Vegas, Nevada, USA. 1999.

[6] Nakajima, T. *Recovery Management in Multiversion Objects*. JAIST Research IS-RR-16S, 1994.

[7] Moss B. "*Nested Transactions – An Approach to Reliable Distributed Computing"*, The MIT Press, 1985.

[8] Wieler, C.A., and Barker, K. "*Reliable and Recoverable Transactions in Object-Based Systems"*. Master Thesis, Manitoba, Canada, 1995.