TRANSACTION MANAGEMENT IN AN OBJECT-ORIENTED DATABASE SYSTEM

Jorge F. Garza, Won Kim

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

ABSTRACT

In this paper, we describe transaction management in ORION, an object-oriented database system. The application environments for which ORION is intended led us to implement the notions of sessions of transactions, and hypothetical transactions (transactions which always abort). The object-oriented data model which ORION implements complicates locking requirements. ORION supports a concurrency control mechanism based on extensions to the current theory of locking, and a transaction recovery mechanism based on conventional logging.

1. INTRODUCTION

In recent years, object-oriented programming [GOLD81, GOLD83, BOBR83, CURR84, SYMB84, LMI85] has gained a tremendous popularity in the design and implementation of a variety of data-intensive application systems. These include artificial intelligence (AI) [STEF86], computer-aided design and manufacturing (CAD/CAM) [AFSA86], and office information systems (OIS) with multi-media documents [IEEE85, AHLS84, WOEL86]. Object-oriented programming offers a number of important advantages for these applications over traditional control-oriented programming. One is the modeling of all conceptual entities with a single concept, namely objects. An object represents anything from a simple number, say, the number 25, to a complex entity such as an automobile or an insurance agency. The state of an object is captured in the instance variables (attributes). The behavior of an object is captured in messages to which an object responds. The messages completely define the semantics of an object. Another advantage of object-oriented programming is the notion of a *class hierarchy (lattice)* and *inheritance* of properties (attributes and messages) along the class hierarchy. The class hierarchy captures the IS-A relationship between a class and its *subclass* (equivalently, a class and its *superclass*). All subclasses of a class inherit all properties defined for the class, and can have additional properties local to them. The notion of property inheritance along the hierarchy facilitates top-down design of the database as well as applications.

In the Advanced Computer Architecture Program at MCC, we have built a prototype object-oriented database system called ORION. ORION directly supports the object-oriented paradigm, adds persistence and sharability to objects through transaction support, and provides various advanced functions that applications from the CAD/CAM, AI, and OIS domains require. Advanced functions supported in ORION include predicate-based queries [BANE88], versions and change notification [CHOU86], composite objects [KIM87], dynamic schema evolution [BANE87b], multimedia data management [WOEL87], and multiple concurrent transaction management.

ORION has been implemented in Common LISP[STEE84] on a Symbolics 3600 LISP machine [SYMB85], and has also been ported to the SUN workstation under the UNIX operating system. The Symbolics version of ORION directly supports data management needs of the PROTEUS expert system shell, also developed in the Advanced Computer Architecture Program at MCC. One of the most useful features of ORION for PROTEUS has turned out to be transaction management.

The objective of this paper is to describe two aspects of transaction management in ORION. First is the enhancements to the performance of transactions for application environments of ORION. Second is the concurrency control and recovery mechanisms used in ORION to support the conventional model of transaction. ORION satisfies the concurrency control requirements of the rich object-oriented data model it supports, in particular, operations on a class lattice and composite objects (which is essentially a part hierarchy for complex objects). The enhancements to the performance of transactions include the concepts of sessions and hypothetical transactions. Both features happen to be important requirements in PROTEUS. A session is a sequence of transactions, and the active transaction of a session may exist simultaneously in multiple windows on a workstation to alleviate the problem of long-duration waits in long-duration transactions in interactive application environments. A hypothetical transaction is a transaction which always aborts, and is important in application environments in which the users often experiment with 'what-if' changes to the database. A hypothetical transaction allows the database system to avoid much of the overhead involved in concurrency control and recovery of normal transactions.

The remainder of this paper is organized as follows. Section 2 provides a brief review of the basic object concepts. In Section 3, we provide an overview of the ORION architecture and the transaction subsystem in ORION. In Section 4 and 5, we discuss, respectively, the concepts of sessions and hypothetical transactions, as implemented in ORION. Sections 6 and 7 describe, respectively, the concurrency control and recovery mechanisms we support in ORION. Section 8 concludes the paper.

2. REVIEW OF OBJECT-ORIENTED CONCEPTS

In this section, we review basic object-oriented concepts that are relevant to our discussions in the remainder of this paper. This section is extracted from our full paper on the ORION data model in [BANE87a].

objects, attributes, methods, and messages

In object-oriented systems, all conceptual entities are modeled as objects. An ordinary integer or string is as much an object as is a complex assembly of parts, such as an aircraft or a submarine. An object consists of some private memory that holds its state. The private memory is made up of the values for a collection of attributes (also called instance variables). The value of an attribute is itself an object, and therefore has its own private memory for its state (i.e., its attributes). A primitive object, such as an integer or a string, has no attributes. It only has a value, which is the object itself. More complex objects contain attributes, through which they reference other objects, which in turn contain attributes.

The behavior of an object is encapsulated in *methods*. Methods consist of code that manipulate or return the state of an object. Methods are a part of the definition of the object. However, methods, as well as attributes, are not visible from outside of the object. Objects can communicate with one another through messages. *Messages* constitute the public interface of an object. For each message understood by an object, there is a corresponding method that executes the message. An object reacts to a message by executing the corresponding method, and returning the state of the object.

classes, class hierarchy, inheritance, and domains

If every object is to carry its own attribute names and its own methods, the amount of information to be specified and stored can become unmanageably large. For this reason, as well as for conceptual simplicity, 'similar' objects are grouped together into a *class*. All objects belonging to the same class are described by the same set of attributes and methods. They all respond to the same messages. Objects that belong to a class are called *instances* of that class. (In this paper, we will use the terms instances and objects interchangeably.) A class describes the form (attributes) of its instances, and the operations (methods) applicable to its instances. Thus, when a message is sent to an instance, the method which implements that message is found in the definition of the class.

Grouping objects into classes helps avoid the specification and storage of much redundant information. The concept of a *class hierarchy* extends this *information hiding* capability one step further. A class hierarchy is a hierarchy of classes in which an edge between a pair of nodes represents the IS-A relationship; that is, the lower level node is a specialization of the higher level node (and conversely, the higher level node is a generalization of the lower level node). For a pair of classes on a class hierarchy, the higher level class is called a *superclass*, and the lower level class a *subclass*. The attributes and methods (collectively called properties) specified for a class are *inherited* (shared) by all its subclasses. Additional properties may be specified for each of the subclasses. A class inherits properties only from its immediate superclass. Since the latter inherits properties from its own superclass, it follows that a class inherits properties from every class in its *superclass chain*.

In object-oriented systems, the *domain* (which corresponds to data type in conventional programming languages) of an attribute is a class. If the domain of an attribute of a class C is the class D, an instance of C may take on as the value for that attribute any instance of the class D or a subclass of D. The domain of an attribute may also be a set of instances of a class D and subclasses of D.

3.OVERVIEW OF TRANSACTION MANAGEMENT IN ORION

Figure 1 shows a high level block diagram of the ORION architecture. The message handler receives all messages sent to the ORION system. The messages include user-defined messages, access messages, and system-defined functions. A *user-defined message* is a message to a method that the user defines and stores in ORION. An *access message* is one that retrieves or updates the value of an attribute of a class. System-defined functions include all ORION functions for schema definition, creation and deletion of instances, transaction management, and so on.

The object subsystem of ORION handles all access to objects in the system. Functions provided by the object subsystem include identifier-based and predicate-based query processing, version management, and multimedia information management.

The storage subsystem provides access to objects on disk. It manages the allocation and deallocation of segments of pages on disk, places objects in the database, searches the database for objects, moves pages of data to and from disk.

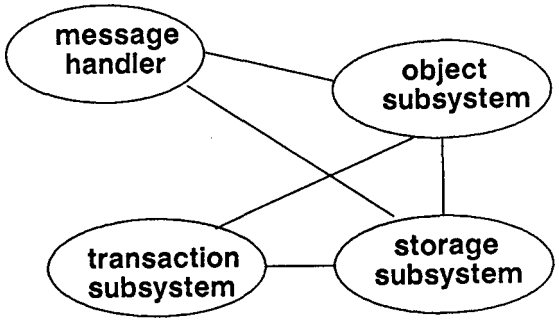

Figure 1. ORION Subsystems

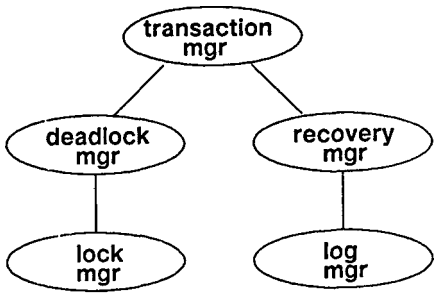

Figure 2. ORION Transaction Subsystem

The transaction subsystem provides mechanisms to protect database integrity while allowing the interleaved execution of multiple concurrent transactions. The transaction subsystem consists of a number of major modules shown in Figure 2. The lock manager is responsible for maintaining the lock table, which indicates the locks each active transaction holds or is waiting for. The storage subsystem interacts with the lock manager to set locks on objects before retrieving or updating the objects. All locks (logical locks) acquired by a transaction are released when the transaction terminates (either commits or aborts).

The deadlock manager is responsible for deadlock detection and resolution. We have implemented a simple technique based on the use of timeouts. Each lock request is given a time limit for lock wait, and, if the request times out, the deadlock manager proceeds to abort the transaction. This technique was used in the System D prototype distributed system [ANDL82].

The log manager accumulates the log of changes to objects (update, create, delete) within a transaction. The log is used to back out a transaction, or to recover from system crashes in the middle of a transaction which leaves the database in an inconsistent state.

The log manager in ORION maintains only the UNDO log (before images) for implementation simplicity. When a transaction commits, the recovery manager instructs the log manager to force the log to disk, and the storage subsystem to force to disk the objects created, updated, or deleted by the transaction.

4. SESSIONS

A user *session* is a sequence of transactions. Since the transactions within a session are strictly serial, that is, one transaction ends before the next one starts, there is only one active transaction per session. As such, the system can use the same data structure for the transaction descriptor for each of the transactions in a session, rather than creating and destroying a transaction descriptor for each transaction. In other words, the concept of session is useful for improving the performance of transactions. Further, the notion of sessions is a hook for eventually supporting the notion of a long-duration transaction as a *set* of transactions, rather than a sequence of transactions [BANC85, KORT87].

A user can use ORION by first creating a session or using the default session the system provides. A *default session* is created immediately upon initialization or restart of ORION. The user may open and close any number of sessions, and multiple windows (or multiple UNIX shells) for the same session (including the default session). All windows of the session run the same transaction, the currently active transaction of the session. Any active transaction of a session will compete for resources (via locking) with those of all other sessions (including the default session).

ORION allows a session to have multiple windows primarily as a short-term solution to the problem of long-duration wait problem that arises in long-duration transactions [HASK82, KIM84]. When multiple sessions are concurrently active, the active transaction of a particular session S may be blocked if it cannot obtain a lock. In such an event, no more requests will be processed from the blocked window until either (1) a certain installation-determined time-out period expires, or (2) the requested lock is obtained. If the user wishes to continue with session S even while waiting for the lock, the user may do so by creating another window for S. Since all windows of a session run the same transaction, they share the same locks; thus a user may create two windows with the same session name, update an object in one window, and examine (or even update) the same object in another window.

The user may close the windows of a session one at a time. If the closed window happens to be the only window of the session, the session itself is closed. If the presently active transaction of that session has not been committed or aborted, ORION automatically aborts that transaction, and then closes the session. Once a window is closed, the most recently deactivated session of that window is reactivated. In other words, on a single window, sessions are activated and deactivated in a stack-like fashion. The user cannot close the default session; it remains open until ORION itself is shutdown.

5. HYPOTHETICAL TRANSACTIONS

ORION supports two types of transactions: normal and hypothetical. A session may contain any sequence of normal and hypothetical transactions. When a normal transaction commits, all its updates are permanently recorded in the database; and when it aborts, all its changes are undone, and irretrievably lost. A *hypothetical transaction*, in contrast, is a transaction that always aborts. No matter how such a transaction is ended, its changes are never reflected in the database. Thus, hypothetical transactions provide a mechanism for experimenting with the effects of 'what-if' changes to the database. Since the changes are never recorded permanently, the user has the freedom of examining the impacts of complex changes to the database, and yet does not have to worry that the database will become corrupted. The concept of a hypothetical transaction is somewhat related to but different from that of a hypothetical database proposed in [STON80, STON81]. A hypothetical database is a database derived from a 'real' database, and persists across any number of 'normal' transactions.

Of course, the conventional transaction mechanism, with its abort option, can be used to provide the desired effect of a hypothetical transaction. However, the conventional transaction mechanism incurs significant overhead to make it possible for a transaction to be recoverable and to shield a transaction from the effects of other concurrently executing transactions. Within a hypothetical transaction, the first time an object is updated, a copy of the object is made for all subsequent updates within the transaction. The initial object is never updated. We call the initial object the *shadow copy*, and the new copy that gets updated the *current copy*. The current copy is discarded when the transaction terminates, regardless of whether the transaction commits or aborts. Further, each hypothetical transaction has its own current copy of an object for updates, so that multiple hypothetical transactions may concurrently update the same object.

Since a hypothetical transaction makes updates only to the current copy of an object, and each hypothetical transaction has its own current copy of the object, only a Share (S) lock needs to be set on the single shadow copy of an object, both for read and update. An S lock is needed, to prevent some concurrently executing non-hypothetical transaction from directly updating the shadow copy of the object, thereby causing the hypothetical transaction to read *dirty data*, data that is subject to a backout by the non-hypothetical transaction.

At a first glance, it seems obvious that the shadow/current copy approach for updates eliminates the need for logging the updates. However, it is somewhat difficult to avoid logging altogether. Although we do not address this issue in the current version of ORION, there is a situation which may require logging of updates of a hypothetical transaction. When the database buffer pool becomes full, some objects will have to be swapped out to make room for new objects that currently executing transactions need. The shadow copies of objects will be the first to be swapped out. After that, if still more space is needed in the buffer pool, some of the current copies will have to go. Then the *current copies* must be logged before getting swapped out and an exclusive (X) lock on the shadow copy, so that the updates are backed out when the transaction terminates.

Logging may be avoided, if we are to simply block the transactions that require new objects. The blocked transactions may resume when some of the current transactions terminate, making it possible swap out their objects. This approach will work, except in the pathological case where the buffer requirement of a single hypothetical transaction exceeds the size of the database buffer pool.

6. CONCURRENCY CONTROL

The ORION transaction subsystem provides a concurrency control mechanism to allow interleaved execution of multiple *concurrent transactions*. Transactions in ORION are *serializable*, which means that ORION completely isolates a transaction from the effects of all other concurrently executing transactions. A serializable transaction acquires a read lock before reading an object, and a write lock before updating an object [GRAY78]. This corresponds to the notion of level-3 consistency in SQL/DS, and protects a transaction from such consistency anomalies as lost updates, dirty read, and un-repeatable read [IBM81]. One reason we have chosen the locking technique for our concurrency control mechanism is that locking is a well-understood technique, having been used in most commercial database systems. Another reason is that the current theory of locking provides a sound basis for

incorporating additional concurrency control requirements imposed by the ORION data model, as we will show in this section.

ORION applications impose locking requirements on three orthogonal types of hierarchy, One is the well-known granularity hierarchy for logical entities, devised to minimize the number of locks to be set. The other two types of hierarchy are consequences of the rich data model that ORION supports, and necessitate extensions to the current theory of locking. One is the class lattice. In object-oriented systems, a class inherits attributes and methods from its superclasses, which in turn inherit these properties from their superclasses. Therefore, while a class and its instances are being accessed, ORION must ensure that the definitions of the class's superclasses (and their superclasses) will not be modified. Another type of hierarchy is the hierarchy of objects called composite object which ORION treats as a unit of physical clustering, enforcement of integrity, and locking. Ideally, we should set one lock for the entire composite object, rather than one on each of the component objects.

We note that, as in conventional commercial database systems, ORION supports both logical locking and physical locking. The lock manager sets a lock on a logical object, namely each node (lockable granule) on a granularity hierarchy, a class lattice, and a composite object. A physical lock is a lock on a physical page that contains logical objects, and must be acquired before logical locks are set on objects in the page. The lock manager holds logical locks until the end of the transaction which acquired them; however, it releases physical locks as soon as objects in the pages are accessed.

In this section, after a brief review of the current theory of locking for the granularity hierarchy, we will describe how ORION uniformly satisfies the locking requirements on all three types hierarchy. To the best of our knowledge, the locking requirements along the class lattice and composite object dimensions have not been reported anywhere.

6.1 REVIEW OF GRANULARITY LOCKING

The fundamental motivation of the granularity locking protocol is to minimize the number of locks to be set in accessing the database [GRAY78, IBM81]. For example, when most of the instances of a class are to be accessed, it makes sense to set one lock for the entire class, rather than one lock for each instance. A lock on a class will imply a lock on each instance of the class. However, when only a few instances of a class need to be accessed, it is better to lock the instances individually, so that other concurrent transactions may access any other instances.

A node of a granularity hierarchy may be locked in one of a number of lock modes. ORION supports the five lock modes defined in [GRAY78]: IS, IX, S, SIX and X. Instance objects are locked only in S or X mode to indicate whether they are to be read or updated, respectively. However, class objects may be locked in any of the five modes. An IS (Intention Share) lock on a class means that instances of the class are to be explicitly locked in S mode as necessary. An IX (Intention Exclusive) lock on a class means instances of the class will be explicitly locked in S or X mode as necessary. An S (Shared) lock on a class means that the class definition is locked in S mode, and all instances of the class are implicitly locked in S mode, and thus are protected from any attempt to update them. An SIX (Shared Intention Exclusive) lock on a class implies that the class definition is locked in S mode, and all instances of the class are implicitly locked in S mode and instances to be updated (by the transaction holding the SIX lock) will be explicitly locked in X mode. An X (Exclusive) lock on a class means that the class definition and all instances of the class may be read or updated. As in SQL/DS [IBM81], an IS, IX, S, or SIX lock on a class implicitly prevents the definition of the class from being updated.

In general, a directed acyclic graph (DAG), such as that shown in Figure 3, is needed to model the *lockable granules* in a database. [GRAY78] presents a locking protocol on a DAG of lockable granules. We summarize it below.

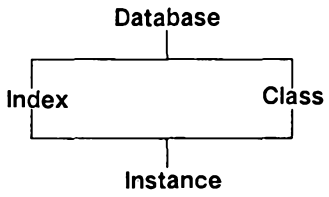

Figure 3. Hierarchy of Lock Granules

(1) To set an explicit S lock on a lockable granule, first set an IS lock on all direct ancestors, along *any one* ancestor chain, of the lockable granule on the DAG.

(2) To set an explicit X lock on a lockable granule, first set an IX or SIX lock on all direct ancestors, along *each* ancestor chain, of the lockable granule on the DAG.

(3) Set all locks in root-to-leaf order.

(4) Release all logical locks in any order at the end of a transaction, or in leaf-to-root order before the end of a transaction

	IS	IX	S	SIX	X
IS	√	√	√	√	No
IX	√	√	No	No	No
S	√	No	√	No	No
SIX	√	No	No	No	No
X	No	No	No	No	No

Figure 4. Compatibility Matrix for Granularity Locking

The compatibility matrix of Figure 4 defines the semantics of the lock modes. A compatibility matrix indicates whether a lock of mode M2 may be granted to a transaction T2, when a lock of mode M1 is presently held by a transaction T1. For example, in Figure 4, we see that when a transaction T1 holds an X lock, no lock of any mode may be granted to any other transaction. However, when T1 holds an S lock, another transaction T2 may be granted an IS or S lock.

6.2 CLASS-LATTICE LOCKING

A number of useful operations can be defined on a class lattice which involve a class and all its subclasses (and their subclasses). We will use the term *class sublattice* to mean a class and all its direct and indirect subclasses on a class lattice. There are two types of operations on a class sublattice: schema changes and queries.

ORION allows about 20 types of dynamic changes to the database schema, without requiring system shutdown or database reorganization [BANE87b]. These include adding or dropping an attribute or method to/from a class, changing the domain of an attribute, changing the inheritance of an attribute or method, adding or dropping a class, adding or dropping a superclass to a class, and so on.

The fact that an object-oriented database schema explicitly captures the IS-A relationship between a pair of classes has two major impacts on the semantics of queries. One is that the search space for a query against a class C may be only the instances of C, or it may encompass the instances of the class sublattice rooted at C. Another major impact is that the domain D of an attribute of a class C is really the class D and all subclasses of D. This means that the search space for a query against a class includes class

sublattices rooted at the domain class of each of its attributes. For example, in Figure 5, an attribute of the class Vehicle is Manufacturer, whose domain is the class Company. The Manufacturer attribute of Vehicle may take on as value an instance of the class Company or an instance of any subclass of Company, for example, AutomobileCompany. Then to evaluate a query against the class Vehicle, it is clear that the search space for the query includes the class sublattice rooted at the class Company.

We have developed (and implemented) two different approaches to satisfying the two types of locking requirements on a class lattice. The approach selected for ORION is to apply the lock modes and locking protocol for a granularity hierarchy to a class lattice. This means that for a query involving a class C and all its descendants, and for a schema change operation on a class C, a lock is set not only on the class C, but also on each of its descendant classes on the class sublattice. The following illustrates this simple protocol.

(1) Select all instances of vehicle and its subclasses such that ...
   (a) lock vehicle class object and those of vehicle's subclasses in IS mode
   (b) lock every selected instance in S mode

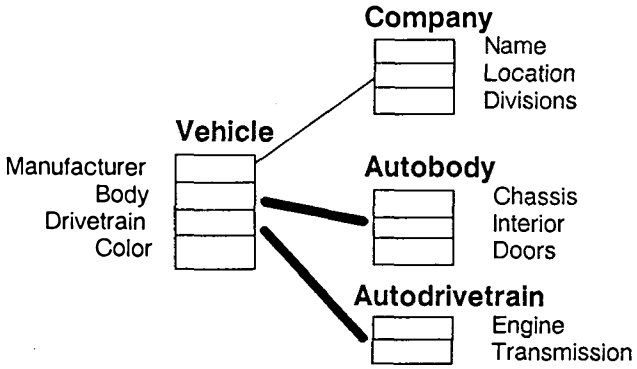

Figure 5. Schema for the Class Vehicle

(2) Change the definition of the vehicle class
   (a) lock vehicle class object and those of vehicle's subclasses in X mode

This approach is particularly appropriate when accessing a class near the leaf level of a deep class lattice. A potential disadvantage of this simple approach is the locking overhead for changes to the definition of a class or for a query against a class, when the class or the domains of the attributes of the class are close to the root of a deep class lattice. We felt that many potential applications of ORION would access objects more frequently via their identifiers than via complex queries involving domain classes, and that they would require changes to the class definitions relatively infrequently; and, as such, we felt that the advantage of this approach outweighes potential disadvantages.

However, a protocol that is efficient for accessing a class near the root of a deep class lattice is clearly useful for an application environment in which complex queries involving a domain class near the root are frequently used. The alternate protocol we developed requires introduction of two new lock modes: read-lattice (R) and write-lattice (W). An R lock on a class C is in essence an explicit S lock on the class C, and implicit S locks on all subclasses of the class C. A W lock on a class is similarly an explicit X lock on the class, and an implicit X lock on each of the subclasses of the class. We will assume that an X lock on a class allows updates to instances of the class, and it allows the definition of the class to be read but not updated. To update the definition of a class, a W lock must be set on the class. Since the R and W modes cause implicit locking of the subclasses of a class sublattice, we need to lock the superclasses of the class sublattice in intention modes IR and IW, respectively. Further, all of the lock modes discussed in the previous subsection will require the IR or IW intention-mode locking of the superclasses of a class being locked. This requirement for intention-mode locks is analogous to the intention-mode locks in granularity locking.

The above protocol is correct for a class hierarchy; but it does not work for a class lattice, in which a class may have more than one superclass. Figure 6 provides an example class lattice, in which a class E has two superclasses, C and G. Suppose a transaction T1 sets an IR lock on A and an R lock on C, implicitly locking subclasses D, E, and K in R mode. Suppose now that a transaction T2 sets an IW lock on F, and a W lock on G. T2 will then implicitly lock the classes E and K in W mode. This means that setting intention-mode locks on superclasses of the class being locked is not sufficient to detect conflicting lock requests from different transactions. Before a W lock can be set on the class G, it must first be determined whether its subclasses E and K may have already been implicitly or explicitly locked by other transactions. To do this will require an upward traversal of the class lattice for each of the superclasses of each of the classes on the class sublattice to be locked. This of course can be very expensive, if the class to be locked has many subclasses and many of the subclasses in turn have many superclasses.

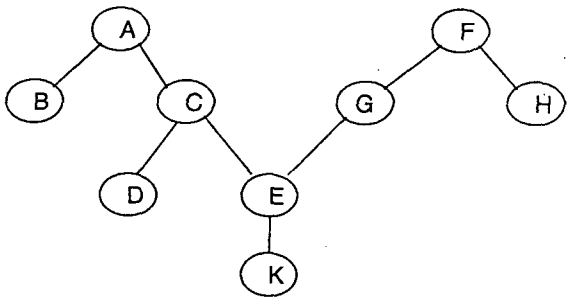

Figure 6. An Example Class Lattice

Our solution to limit the above overhead, when setting an explicit R or W lock on a class C, is to set explicit R or W locks, respectively, on those subclasses of C which have more than one superclass. Then only those subclasses need to be examined for conflict. In the current example, this means that when transaction T1 sets an R lock on C, an explicit R lock will also be set on the class E. Then transaction T2's request for a W lock on the class G can be rejected by simply determining that the class E, the only subclass of G with more than one superclass, is already locked in R mode by transaction T1.

One important point about this protocol is that to set an explicit R or W lock on a class C requires IR or IW locks, respectively, on only one superclass chain of the class C. Further, any superclass chain of the class C may be chosen for locking. For example, the class E in Figure 6 has two superclass chains, A-C and F-G. Only one of them needs to be selected, and the classes along the chain locked in IR or IW mode. The reason this is sufficient is that, unlike the conventional granularity locking protocol, the present protocol does not rely entirely on the intention locks on superclasses for lock conflict resolution. Lock conflict resolution in the protocol requires examination of the locking status of the subclasses of a class being locked. Since the protocol does not rely solely on intention locks for conflict resolution, we need to set the IR or IW intention locks only on the classes along one superclass chain of the class being locked.

The following illustrates the protocol.

(1) Select all instances of vehicle such that ....
   (a) lock vehicle class object in IS mode and those of

vehicle's superclasses along any one superclass chain in IR mode
(b) lock each instance of vehicle in S mode

(2) Select all instances of vehicle and its subclasses
(a) lock vehicle class object in R mode and those of vehicle's superclasses along any one superclass chain in IR mode
(b) lock each subclass of vehicle with more than one superclass in R mode

(3) Select all instances of vehicle and its subclasses such that ....
(a) lock vehicle class object in IS mode and those of vehicle's superclasses along any one superclass chain in IR mode
(b) lock each subclass of vehicle in IS mode
(c) lock each instance of vehicle or any subclass of vehicle in S mode

(4) Change the definition of the vehicle class
(a) lock vehicle class object in W mode, and those of vehicle's superclasses along one superclass chain in IW mode
(b) lock each subclass of vehicle with more than one superclass in W mode.

One potential disadvantage of this protocol is that, even for access requests involving only a single class, it requires intention locks on the superclasses along a superclass chain of the class. In other words, even when a class C is to be locked in IS, IX, S, or X mode, its superclasses along one superclass chain must be locked in IR or IW mode. This is especially undesirable, if the class C is close to the leaf level of a deep class lattice, and access to a single class is much more frequent than access involving a class sublattice.

6.3 COMPOSITE OBJECT LOCKING

Many applications require the ability to define and manipulate a set of objects as a single logical entity for purposes of semantic integrity, and efficient storage and retrieval [LORI83, IEEE85, KIM87]. A composite object is a heterogeneous collection of objects, and we refer to the hierarchy of classes to which the objects belong as a *composite object hierarchy*. The notions of attributes, domains, and object identifier, although powerful, cannot represent the IS-PART-OF relationship between objects. The IS-PART-OF relationship captures the notion that an object *is a part of* another object; and, along with the IS-A relationship, is one of the fundamental data modeling concepts. We note that a composite object is that part of a conventional nested object on which we impose the IS-PART-OF relationship. For example, in the definition of a Vehicle object in Figure 5, only that part of the object which is defined in terms of the attributes (indicated with solid lines) Body and Drivetrain (and their nested attributes) constitute a composite object; the attributes Color and Manufacturer (and its nested attributes) are not part of the Vehicle composite object hierarchy.

A composite object may be implemented in many different ways [LORI83, KIM87]. One aspect of composite object implementation that impacts the locking protocol is whether each dependent object of a composite object contains the identifier of the composite object, that is, the root object of the composite object. In ORION, the components of a composite object do not carry the identifier of the composite object.

The granularity locking protocol does not recognize a composite object as a single lockable granule, like a class or an instance of a class. To lock a composite object using the granularity locking protocol will mean either locking all component classes on a composite object hierarchy, or locking all constituent objects within a composite object. Neither option is satisfactory: the first option will result in the locking of all composite objects that belong to the composite object hierarchy; and the second option can result in a large number of locks.

A locking protocol for composite objects must recognize a composite object as a lockable granule. An intuitive, but incorrect, protocol is as follows. To lock an entire composite object, the root class is locked in IS, IX, S, SIX or X mode; and, if the root class is locked in IS, S, IX, or SIX mode, the root object of the composite object is locked in S, S, X, or X mode, respectively. Further, the component classes are also locked in IS, IX, S, SIX, or X mode, respectively. However, the dependent objects of a composite object are only implicitly locked; that is, they are not locked.

To lock a class, which is a component class on a composite object hierarchy, independently of the composite object hierarchy, e.g., to find all oval-shaped doors, the class should be locked in IS, IX, S, SIX or X mode.

This protocol is not quite correct. Suppose a transaction T1 has set an X lock on a composite object. This means that an IX lock has been set on each of the component classes of the composite object hierarchy. Then another transaction T2 attempts to update some instances of one of the component classes independently of the composite object hierarchy. T2 will set an IX lock on the class, and then X locks on instances of the class to be updated. However, the identifier for the root object of a composite object is not stored in any of the dependent objects, and there is no way to prevent the transaction T2 from setting X locks on those dependent objects of the composite object that the transaction T1 has implicitly locked in X mode.

We need three new lock modes to make the protocol correct: ISO, IXO, SIXO, corresponding to the IS, IX, and SIX modes, respectively. The compatibility matrix of Figure 7 completely defines the semantics of all lock modes we have to support granularity locking, class lattice locking, and composite object locking. As shown in the compatibility matrix, the ISO mode conflicts with the IX mode, and IXO and SIXO modes conflict with both the IS and IX modes. To lock a composite object, the root class is locked, as before, in IS, IX, S, SIX, or X mode. However, each of the component classes of the composite object hierarchy is now locked in ISO, IXO, S, SIXO, or X mode, respectively.

current mode \ requested mode	IS	IX	S	SIX	X	ISO	IXO	SIXO
IS	√	√	√	√	No	√	No	No
IX	√	√	No	No	No	No	No	No
S	√	No	√	No	No	√	No	No
SIX	√	No	No	No	No	No	No	No
X	No	No	No	No	No	No	No	No
ISO	√	No	√	No	No	√	√	√
IXO	No	No	No	No	No	√	√	No
SIXO	No	No	No	No	No	√	No	No

Figure 7. Compatibility Matrix for Granularity Locking and Composite Object Locking

This protocol allows multiple users to read and update composite objects having the same root class, as long as they update different composite objects. However, if there is even one reader via the composite object hierarchy, there can be no direct updaters of instances of component classes. Further, if there is even one direct reader of instances of a component class via the class lattice, there can be no updaters via the composite object hierarchy.

The following illustrates our locking protocol for accessing composite objects, and for directly accessing instances of a component class of a composite object hierarchy.

(1) Select vehicle composite objects such that ...
   (a) lock vehicle class object in IS mode
   (b) lock the selected instances of vehicle in S mode
   (c) lock the component class objects in ISO mode

(2) Update all vehicles or their components such that ...
   (a) lock vehicle class object in IX mode
   (b) lock the selected instances of vehicle in X mode
   (c) lock the component class objects in IXO mode

(3) Select all doors such that ...
   (a) lock the door class object in IS mode
   (b) lock the selected door instances in S mode

(4) Update all doors such that ...
   (a) lock the door class object in IX mode
   (b) lock the selected door instances in X mode

6.4 LOCK CONVERSION

In the previous three subsections, we described the lock modes and locking protocol that satisfies the locking requirements on three types of hierarchy in an object-oriented database. In this subsection, we complete our locking protocol by describing the conversion of lock modes.

A transaction sometimes needs to trade in a lock it currently holds on an object and acquire a lock in a more exclusive mode, that is, to convert a lock from one mode to a more exclusive mode. There are two reasons for lock conversion. One is to increase concurrency. For example, suppose that a transaction is to read a composite object from the database, perform extensive computations on it, and then write it back. It is reasonable for the transaction to set an S lock on the composite object when it reads it, and then to request an X lock when it attempts to write it out. If, instead, the transaction sets an X lock on the composite object when it reads the object, it will block all other transactions from reading the object during the extensive computation phase of the transaction. Another reason for lock conversion is to offload some of the bookkeeping from the callers of the lock manager. If the caller is to avoid redundant requests for locks on objects, it must keep track of all the locks it has acquired.

In Figure 8, we provide a lock conversion matrix for the lock modes supported in ORION. Each entry in the matrix, corresponding to the i-th element in the vector of current lock modes and to the j-th element in the vector of requested lock modes, indicates the new lock mode that can be granted to a transaction. For example, if a transaction currently holds an IX lock on an object, and requests an S lock on the object, the transaction will be granted an SIX mode lock.

6.5 INDEX-PAGE LOCKING

ORION supports B+-tree indexing [COME79] on attributes of a class that the user specifies to speed up associative searches of objects that satisfy an arbitrary combination of predicates. The locking technique we use on index entries is not new; it has been adopted from [BAYE77, GUIB78, BILI85]. We summarize it here only for completeness.

current mode \ requested mode	IS	IX	S	SIX	X	ISO	IXO	SIXO
IS	IS	IX	S	SIX	X	S	SIXO	SIXO
IX	IX	IX	SIX	SIX	X	X	X	X
S	S	SIX	S	SIX	X	X	X	X
SIX	SIX	SIX	SIX	SIX	X	SIXO	SIXO	SIXO
X	X	X	X	X	X	X	X	X
ISO	S	X	S	SIXO	X	ISO	IXO	SIXO
IXO	SIXO	X	SIXO	SIXO	X	IXO	IXO	SIXO
SIXO	SIXO	X	SIXO	SIXO	X	SIXO	SIXO	SIXO

Figure 8. Lock Conversion Matrix

1. To fetch an index page for a read-only access to an index, acquire a page-lock in Share (S) mode on the index page.
2. To insert, delete, or update any entries in an index page, acquire a page-lock in Exclusive (X) mode on the page. An X lock on a non-leaf index page should be released as soon as it is not needed.
3. In the event of an index-page split during insertion of new entries, acquire an X lock on both the old index page and the newly allocated index page.
4. In the case of a page-merging during deletion of entries, set X locks on both the old index page that has become empty and the index page into which entries from the old page have been distributed.

All locks on index pages, except the X locks on the leaf pages of an index, should be released as soon as the pages are no longer needed. The X locks on the leaf pages must be held until the end of the transaction, when the transaction manager releases them, along with all other locks acquired during the transaction.

A deadlock may result if a transaction reads an index page P, releases the S lock on P (thereby allowing another transaction to lock the page), and then must acquire an X lock on P because of the split/merge of index pages at a lower level of the B+ tree that percolates upward to P. To prevent deadlocks due to such upward percolation of page split/merge, we use a special technique [GUIB78], which changes the nature of a B+ tree somewhat. During insertion, while accessing an index page P (holding an X lock on it), the index manager must determine if the insertion will result in the split of Q, the next page to be accessed (a child page of P); if so, it will split Q, into Q and R, and insert an entry in P that will point to the newly allocated page R. Conversely, during deletion, while accessing an index page P (with an X lock on it), the index manager will determine if a pair of child pages of P (the next page to be accessed and one to its left or right) may be merged; if so, it moves entries from one to the other. In both cases, entries in the parent page that point to the child pages are updated while the parent page is locked in X mode. That is, the transaction does not attempt to re-acquire the X lock on the parent page, after it has released a lock on it.

7. RECOVERY

Most commercial database systems support database recovery from soft crashes (which leave the contents of disk intact) and hard crashes (which destroy the contents of disk). ORION supports transaction recovery only from soft crashes and user-initiated transaction abort. In other words, ORION does not support archival dumping of the database to recover from disk head crashes. The need to support multiple concurrent transactions led us to select the logging scheme over the shadow-page scheme [GRAY81]. There are three options in a log-based transaction recovery scheme: maintain only the UNDO log, only the REDO log, or both the UNDO and REDO logs. We have implemented the UNDO logging option, for implementation simplicity. This approach requires the pages containing updated objects to be forced to disk at the end of a transaction: this is necessary because if the updates only remain in the buffer pool and the system crashes, there is no way to re-do them using the UNDO log.

The use of logging in ORION for multimedia data and index page recovery is somewhat interesting, and will be described in the remainder of this section.

multimedia data logging

[HASK82] pointed out that a naive use of the logging techniques can cause some serious problems for the long,

multimedia data that ORION must support. A strict log-based recovery will keep UNDO and/or REDO logs of very long data. As shown in Figure 9, we distinguish a multimedia data from its descriptor. The descriptor references, via an object identifier, the multimedia data. The storage subsystem maintains a free list of storage blocks which may be allocated for storing multimedia data. The long data manager in the object subsystem logs the changes in the descriptors and the free list, but not the multimedia data. In this way, if a transaction that created a multimedia data aborts, the descriptor will be returned to referencing nil, and the entry in the free list which points to the storage block allocated to the multimedia data will be reset as available for allocation. Similarly, if a transaction that deleted a multimedia data aborts, the reference in the descriptor will be returned to its initial value and the free-list entry will be deleted. This technique, proposed in [HASK82],

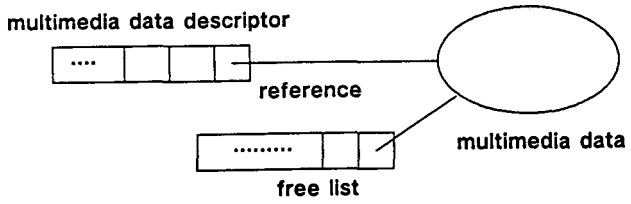

Figure 9. Logging of Multimedia Data

achieves the shadowing effect, without the complexity associated with the conventional shadowing mechanism [LORI77, GRAY81].

index page logging

In some commercial database systems, such as SQL/DS, updates to index pages are not logged [GRAY81]. This is to reduce the logging overhead during normal transaction processing. However, it complicates the recovery manager, since the recovery manager must reconstruct the entries in the index pages from the log of updates to the data pages. We realize that this is the right approach; however, to simplify our implementation, we log updates to index entries. ORION implements the following protocol for index-page logging.

1. In the case of an index-page split during insertion, it will distribute half of the entries in the current page to a new page. The entry in the parent page that points to the new page must be logged. Also, those entries that are deleted from the current index page must be logged; however, the entries that are copied into the new page need not be logged. In case of a crash, the entire new page is simply dropped.
2. In the case of a index-page merge during deletion, entries in the current page are copied over to another page. The entry in the parent page that points to the current page must be logged. Further, the entries that are copied into the new page must be logged; however, the entries being deleted from the current page need not be logged. In case of a crash, the current page will be re-claimed, since a dropped page is not returned to a free-page list until the transaction commits.

The index-page entries that are logged must be identifiable through the page id and an offset within the page. The operation code (insert or delete) and the length of the entry must be recorded as well.

8. CONCLUDING REMARKS

ORION is an object-oriented database system which we have prototyped in the Advanced Computer Architecture Program at MCC. ORION is intended for application environments in such domains as artificial intelligence, computer-aided design, and office information management with multimedia documents. This paper described two aspects of transaction management in ORION. First is the enhancements to the performance of transactions for application environments of ORION. Second is the concurrency control and recovery mechanisms used in ORION to support the conventional model of transaction.

The enhancements to the performance of transactions include the concepts of sessions and hypothetical transactions. A session is a sequence of transactions which defines a larger unit of work than a single transaction. The concept of a session allows the system to re-use the data structures which the system maintains for a sequence of related transactions. The active transaction of a session may exist simultaneously in multiple windows on a workstation. This is a short-term solution to the problem of long-duration waits in long-duration transactions in interactive application environments. A hypothetical transaction always aborts, and is an important mode of interaction against the database in application environments in which the users experiment with 'what-if' changes to the database. ORION uses hypothetical transactions to avoid much of the logging overhead and waits caused by exclusive locks on objects in normal transactions.

The object-oriented data model which ORION supports has significantly complicated the concurrency control requirements for ORION transaction management. In an object-oriented database, three types of hierarchy locking are necessary. One is the conventional granularity-hierarchy locking, which is useful for minimizing the number of locks to be set. Another is the class-lattice locking, which is needed to allow access to instances of a class while preventing changes to the definitions of the superclasses of the class. Another is the composite object locking to make it possible to set only one lock on a composite object, rather than all its component objects.

ACKNOWLEDGEMENTS

Hong-Tai Chou helped us in the discussions of locking and logging of index pages. Nat Ballou pointed out the need to support hypothetical transactions.

REFERENCES

[AFSA86] Afsarmanesh, H., Knapp, D., McLeod, D., and Parker, A. "An Object-Oriented Approach to VLSI/CAD," in *Proc. Intl Conf. on Very Large Data Bases*, August 1985, Stockholm, Sweden.

[AHLS84] Ahlsen M., A. Bjornerstedt, S. Britts, C. Hulten, and L. Soderlund. "An Architecture for Object Management in OIS," *ACM Trans. on Office Information Systems*, vol. 2, no. 3, July 1984, pp. 173-196.

[ANDL82] S. Andler, et al. "System D: A Distributed System for High Availability," in *Proc. 8th Intl Conf. on Very Large Data Bases*, Oct. 1982.

[ATWO85] Atwood, T.M. "An Object-Oriented DBMS for Design Support Applications," *Proc. IEEE COMPINT 85*, Montreal, Canada, pp. 299-307.

[BANC85] Bancilhon, F., W. Kim, and H. Korth. "A Model of CAD Transactions," in *Proc. Intl Conf. on Very Large Data Bases*, August 1985, Stockholm, Sweden.

[BANE87a] Banerjee, J., et al. "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems*, Jan. 1987.

[BANE87b] Banerjee, J., W. Kim, H.J. Kim, and H.F. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proc. ACM SIGMOD Conf. on the Management of Data*, San Francisco, Calif., May 1987.

[BANE88] Banerjee, J., W. Kim, K.C. Kim. "Queries in Object-Oriented Databases," to appear in *Proc. Intl Conf. on Data Engineering*, Los Angeles, Calif., Feb. 1988.

[BAYE77] Bayer, R., and M. Schkolnick. "Concurrency of Operations on B-trees," *Acta Informatica*, vol. 9. pp. 1-21, 1977.

[BILI85] Biliris, A. "Concurrency Control on Database Indexes: the mU Protocol," Technical Report: #85/014, Department of Computer Science, Boston University, December 1985.

[BOBR83] Bobrow, D.G.. and M. Stefik. *The LOOPS Manual*, Xerox PARC, Palo Alto, CA., 1983.

[CHOU86] Hong-Tai Chou and Won Kim. "A Unifying Framework for Version Control in a CAD Environment," in *Proc. 12th Intl Conf. on Very Large Data Bases*, August. 1986.

[CHRI84] Christodoulakis, S., et al. "Development of a Multimedia Information System for an Office Environment," in *Proc. Intl Conf. on Very Large Data Bases*, Singapore, 1984, pp. 261-271.

[COME79] Comer, D. "The Ubiquitous B-Tree," *Computing Surveys*, Vol. 11 No. 2, June 1979.

[COPE84] Copeland, G. and D. Maier. "Making Smalltalk a Database System,"in *Proc. ACM SIGMOD Intl Conf. on the Management of Data*, June 1984, pp. 316-325.

[CURR84] Curry, G.A. and R.M. Ayers. "Experience with Traits in the Xerox Star Workstation," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 519-527.

[GOLD81] Goldberg, A. "Introducing the Smalltalk-80 System," *Byte*, vol. 6, no. 8, August 1981, pp. 14-26.

[GOLD83] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA 1983.

[GRAY78] Gray, J.N. *Notes on Data Base Operating Systems*, IBM Research Report: RJ2188, IBM Research, San Jose, Calif. 1978.

[GRAY81] J.N. Gray, et al. "The Recovery Manager of a Data Management System," *ACM Computing Surveys, vol. 13, no. 2*, June 1981, pp. 223-242.

[GUIB78] Guibas, C., and R. Sedgewick. "A Dichromatic Framework for Balanced Trees," in *Proc. 19th Annual Symposium on Foundation of Computer Science*, 1978.

[HASK82] Haskin, R. and R. Lorie. "On Extending the Functions of a Relational Database System," in *Proc. ACM SIGMOD Conf.*, June 1982, pp. 207-212.

[IBM81] SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corporation, Jan. 1981.

[IEEE85] *Database Engineering*, IEEE Computer Society, vol. 8, no. 4, December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky).

[KIM84] Kim, W., D. McNabb, R. Lorie, and W. Plouffe. "A Transaction Mechanism for Engineering Design Databases," in *Proc. Intl. Conf. on Very Large Databases*, 1984, Singapore.

[KIM87] Kim, W., et al. "Composite Object Support in an Object-Oriented Database System," in *Proc. Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1987, Orlando, Florida.

[KORT87] Korth, H., W. Kim, and F. Bancilhon. "On Long-Duration CAD Transactions," to appear in *Information Science* in 1987.

[LMI85] *ObjectLISP User Manual*, LMI, Cambridge, MA, 1985.

[LORI77] Lorie, R. "Physical Integrity in a Large Segmented Database," *ACM Trans. on Database Systems*, vol. 2, no. 1, pp. 91-104, March 1977

[LORI83] Lorie, R. and W. Plouffe. "Complex Objects and Their Use in Design Transactions," in *Proc. ACM Database Week: Engineering Design Applications*, May 1983, pp. 115-121.

[STEE84] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb, "Common Lisp", *Digital Press*, 1984.

[STEF86] Stefik, M. and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986, pp. 40-62.

[STON80] Stonebraker, M., and K. Keller. "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System," in *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 58-66, Santa Monica, Calif. 1980.

[STON81] Stonebraker, M. "Hypothetical Data Bases as Views," in *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 224-229, 1981.

[SYMB84] Symbolics, Inc. *FLAV Objects, Message Passing, and Flavors*, Cambridge, MA, 1984.

[SYMB85] Symbolics Inc., "User's Guide to Symbolics Computers," *Symbolics Manual #* 996015, March 1985.

[WOEL86] Woelk, D. W. Kim and W. Luther. "An Object-Oriented Approach to Multimedia Databases," in *Proc. ACM SIGMOD Conf. on the Management of Data*, Washington D.C., May 1986.

[WOEL87] Woelk, D., and W. Kim. "Multimedia Information Management in an Object-Oriented Database System," in *Proc. Very Large Data Bases*, Brighton, England, Sept. 1987.