A CONCURRENCY CONTROL SCHEME FOR CAD TRANSACTIONS

Henry F. Korth and Won Kim

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-34 December 1985

A Concurrency Control Scheme for CAD Transactions

Henry F. Korth
Department of Computer Sciences
University of Texas
Austin, Texas 78712-1188

Won Kim
Microelectronics and Computer Technology Corporation
9430 Research Blvd.
Austin, Texas 78759

ABSTRACT

A CAD (Computer-Aided Design) environment requires a significantly different model of transaction from that developed for typical data-processing transactions. One reason is that a CAD database system must support CAD objects as units of retrieval and update. Another reason is that CAD transactions are of longer duration than standard transactions. In this paper, we present a concurrency control scheme which supports CAD objects and takes advantage of the CAD transaction model we proposed earlier to achieve greater parallelism. We also outline techniques for implementing the concurrency control and recovery manager of our CAD transaction model.

1. Introduction

The goal of any concurrency control scheme for a database system is to allow for a high degree of parallelism among the members of a set of concurrently executing transactions, while regulating that parallelism so as to ensure that the consistency of the data is preserved. In traditional data-processing style applications, concurrency control theory is based on the notion of *serializability*. A concurrent execution of a set of transactions is said to be *serializable* if it is equivalent to some *serial* execution of that set of transactions.

However, serializability is a strong requirement. In systems in which locking is used as the concurrency control mechanism, it is necessary to use *two-phase locking* to ensure serializability unless some structure is imposed on the data or additional semantic information about the data is made available to the concurrency control mechanism. A drawback to two-phase locking is that it may require locks to be held for a substantial fraction of a transaction's duration, thus restricting the amount of parallelism.

In a CAD environment, transactions are often of long duration. This property exacerbates the disadvantages of two-phase locking. In [BANC85], we proposed a transaction model for CAD transactions in which serializability is not required. Instead, a consistency constraint is stated for the database and an invariant is defined for each transaction. Executions that preserve the requisite invariants are allowed, including some executions that may not be serializable. The model allows for transactions to be nested within other transactions [MOSS81]. Nesting allows for the modeling of collections of *cooperating* transactions that represent a group of designers that are collaborating closely on a design. Other types of interaction among transactions may be represented also, such as a *client/subcontractor* interaction, in which a subtransaction may run

concurrently with its parent, but is logically equivalent to a procedure call by the parent.

In [BANC85], we presented a few simple schemes for concurrency control within the model. In this paper, we present an approach to concurrency control that is designed especially for this CAD transaction model. First, we extend the granularity DAG of [GRAY76] to represent versions of CAD objects. We then extend the lock modes of [KORT83] to include the special database operations associated with CAD transactions and versions. Because we do not use serializability as our notion of correctness, we define new criteria for the correctness of a database concurrency scheme. We show that our definitions are compatible with the CAD transaction model of [BANC85] and prove that our scheme is correct under our new definition. We also show techniques for implementing the concurrency control and recovery manager for our model of transactions.

2. Extending the Multiple-Granularity DAG

In this section, we review the directed acyclic graph (DAG) used by [GRAY76] and [KORT83] to represent multiple granularities of data. We then show how this approach can be extended to represent versions.

2.1. Standard Granularities

Multiple-granularity locking is motivated by the fact the different transactions require different units of data. Some may need a few records chosen randomly from a relation; others may need a whole relation, etc. In a design database, design objects may consist of tuples stored in several distinct relations. Many transactions will access data in object units rather than in relation or tuple units. It is possible to implement a locking scheme using only one lock granularity. However, such a scheme imposes inefficiencies on the system:

- Transactions that access data in large units (e.g., design objects or relations) need to take a large number of locks. This increases the amount of overhead imposed by locking.
- Transactions that access data in small units (e.g., records) may have to lock a larger unit of data than is actually needed. This has the effect of reducing the amount of potential concurrency in the system.

Multiple-granularity locking is a technique that allows a transaction to lock data using a granule size that corresponds closely to that with which the transactions accesses data.

We describe a collection of lock granularities by defining a *granularity scheme,* which specifies the types of granularity (e.g., record, file, relation) that we shall allow. A granularity scheme also gives a sub-granule relationship between pairs of granularities (e.g., record is a sub-granule of file). Typically, we represent a granularity scheme by a directed graph. Figure 1 shows a granularity scheme for a simple data-processing database.

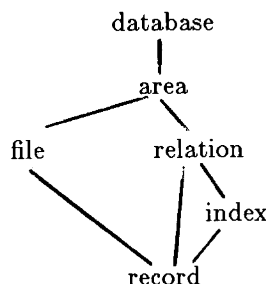

Figure. 1 Granularity DAG for a Conventional Database

The database is partitioned into a collection of areas. Areas are partitioned in several ways:

- An area contains a collection of files. Although every record is contained in some file, it may be the case that records of a particular relation are spread over many files.

- An area contains a collection of database relations. Records of a relation may appear in several different files.

Having defined a granularity scheme, we may construct an *instance* of this scheme for any given instance of the database. An instance is a directed acyclic graph (DAG) that shows the sub-granules and super-granules for each granule of data in the database. There is a one-to-one correspondence between the nodes of the DAG and the granules of data in the instance. An edge (a,b) appears in the DAG if and only if the granule associated with node b is a subgranule of the granule associated with node a. Thus, the leaves of the DAG represent the smallest units of data at which we are willing to allow locking. The multiple-granularity locking protocol uses the granularity-scheme instance.

The directed-graph for an instance is necessarily acyclic, since it represents the notion of sub-granule. Since the granule representing the entire database contains all other granules as sub-granules, the database granule serves as the root of this DAG. Figure 2 shows a sample instance for the scheme of Figure 1.

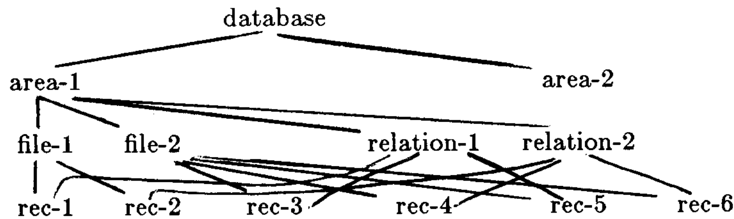

Figure. 2 Instance of Granularity Scheme of Figure 1.

We review briefly the locking protocol of [GRAY76] using the instance of Figure 2 as our example. The database is split into 2 areas. Area-1 holds 2 relations (rel-1 and rel-2), and 2 files (file-1 and file-2). The records contained within the relations, and files of area-1 and records rec-1, rec-2, ..., rec-n. We have not shown the structure of area-2 in our figure due to space considerations.

We use the DAG of Figure 2 to explain the semantics of locking in a multiple-granularity scheme. A lock on file file-1, for example, locks all the records within file-1, but does so with only one lock request. In such a case, we say that the records of file-1 (rec-1 and rec-2) are locked *implicitly*. Similarly, a lock on area-1 locks all the files, and relations in area-1 implicitly, and thus locks all records contained in area-1 implicitly.

The astute reader will note what appears to be a flaw in this scheme. One transaction, say t_1 may lock rec-1. Meanwhile, transaction t_2 may lock area-1. The result is that both transactions have locked rec-1 and we apparently have defeated the entire purpose of locking. This potential flaw is dealt with using *intention* locks. We shall present this form of lock in Section 4. Another apparent flaw is that two transactions may lock record-1, one by locking file-1 and the other by locking relation-1. This issue, too, is dealt with in Section 4 where the semantics of implicit locking are defined precisely.

2.2. Design-Object Granularity

The granularity scheme of Figure 1 needs to be augmented in order to describe granularities appropriate for CAD transactions. We need to include a granularity that represents *design objects*. A design object consists of records from several relations and several files [HASK82]. Thus, the design-object granularity is neither a sub-granule nor a super-granule of either the file or record granularities. This leads to a granularity scheme as shown in Figure 3.

The scheme of Figure 3 is not sufficiently general to represent design hierarchies of composite objects. In general, an object may contain other objects. That is, objects may be composite. Sub-objects may be shared among several objects. We do not know, in general, how many sub-objects an object might have in a particular instance, nor is there any fixed bound on the depth

of nesting. Figure 4 shows an example of a nesting of objects. (Note that we have omitted the structure of area2 and object2 to simplify the figure. Also, we have not shown the file granularity.)

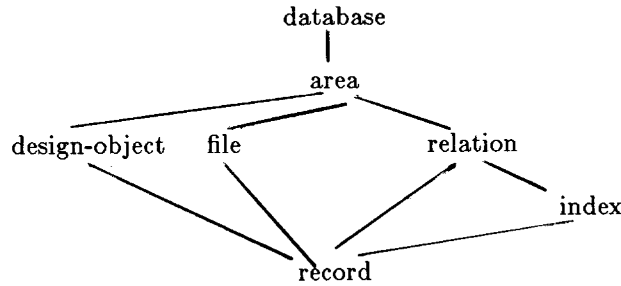

Figure 3 Granularity Scheme for a CAD database (without composite objects)

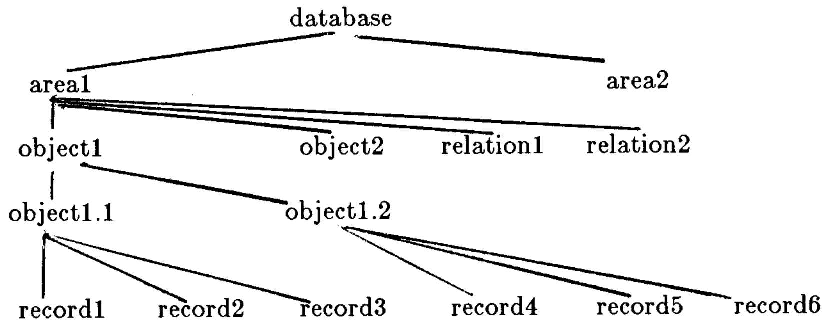

Figure. 4 Instance of Granularity Scheme with Composite Objects.

In order to represent composite objects, we add an edge in the directed graph from a design object to itself, thus creating a cycle. This edge indicates that an object granule may be a subgranule of another object granule. Although the resulting granularity scheme is cyclic, we restrict *instances* of granularity scheme to be acyclic. This does not constrain our model in any practical sense, since our condition of acyclicity simply requires that no design object contain itself. Figure 5 shows our granularity scheme for composite objects. Figure 4 is, in fact, a sample instance of the granularity scheme of Figure 5.

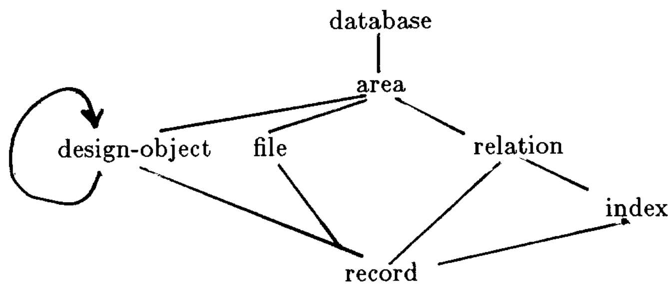

Figure 5 Granularity Scheme for a CAD database (with composite objects)

2.3. Versions of Design Objects

We consider the question of allowing multiple *versions* of data [KATZ84]. The idea of versions has been used previously in concurrency-control schemes. In [BERN81], versions are used in conjunction with timestamp-ordering for concurrency control. Versions in schemes such as those in [BERN81] are not visible to users and exist solely to assist in concurrency control. In a CAD

environment, versions are a natural consequence of the design process. Versions may represent released designs as well as modifications of existing designs. Thus, the distinction among versions must be visible to the user.

In a practical system, one may wish to place an upper bound on the number of versions allowed for a given data item. However, no matter what bound we choose, [PAPA82] has shown that raising that bound by 1 will allow still greater potential parallelism.

For these reasons, we do not set an a priori bound on the number of versions and seek an approach that will accommodate however many versions we are willing to allocate space for. In general, we can represent versions in an instance of our granularity scheme as follows: Let n be a node of the DAG. If n represents a data granule for which multiple versions exist, create a node n_i for each version v_i of the data. Add an edge (n, n_i) for each node n_i. Create a copy of the subtree of node n for each version and associate one copy with each of the n_i. Mark node n as representing a multiversion granule.

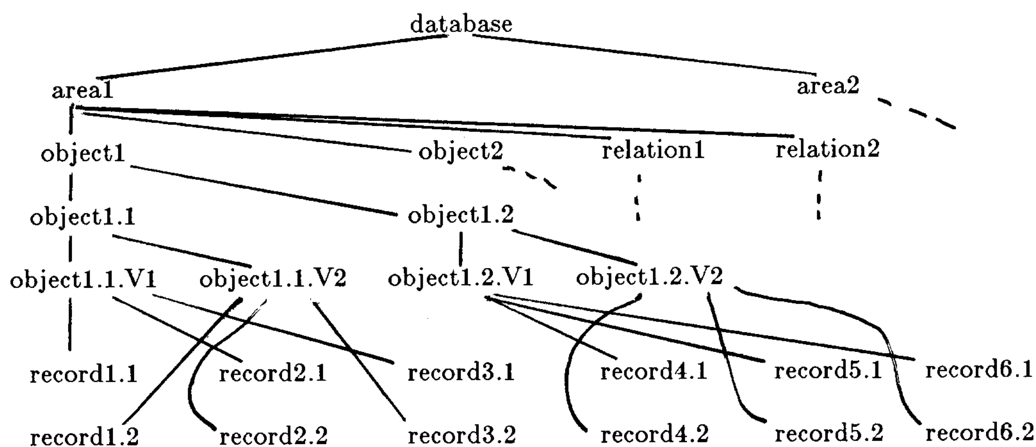

Figure. 6 Instance of Granularity Scheme with Composite Objects and Multiple Versions.

Thus, we have a two-level scheme for representing a multiversion datum in a DAG. The higher level represents the datum itself (all its versions). The lower level represents the individual versions. In order to model a practical CAD environment, it is sufficient to allow versions only for design objects. However, we do not need to make this restriction a requirement of our model.

Figure 6 shows the instance of Figure 4 extended to include several versions of the design objects.

3. Extended Lock Modes for CAD Databases

There are several types of access to CAD data that may be required. Often, design data (design file) is "checked out" of a design database, operated on by the transaction in a designer's private database, and then "checked in", perhaps as a new version [KATZ84]. Below, we list four lock modes and show intuitively how these modes can be used to provide access to CAD data under different requirements for transaction isolation.

- X -- Exclusive
- W -- Write
- R -- Read
- D -- Read dirty snapshot

Figure 7 shows a *lock compatibility matrix* for our set of lock modes. To determine if a lock in mode a can be granted to a transaction despite another transaction already holding a b mode lock on the same datum, we look in the row for a and the column for b. If the entry is *true* then the request is allowed. Otherwise the lock request is denied.

	X	W	R	D
X	*false*	*false*	*false*	*false*
W	*false*	*false*	*false*	*true*
R	*false*	*false*	*true*	*true*
D	*false*	*true*	*true*	*true*

Figure 7: Compatibility Matrix

We now show to which of several modes of access to a CAD database these lock modes correspond. In what follows, for a data item n for which versions exist, we shall use n_i to denote the node corresponding to a particular version. We assume that if $i < j$, version n_i is older than version n_j.

- Protect all versions: This means obtaining exclusive access to all of the n_is of a version granule n. This is accomplished by locking n in X mode. Since this lock provides an implicit X lock on all versions, no concurrent access to any version is allowed. An X lock on n also precludes the creation of a new version by another transaction, since a new version becomes a child of n.
- Create new version: When a new version of a datum is created, a node corresponding to it must be inserted into the granularity DAG. The creator of a new version needs write access to that version (or exclusive access). Thus, creation of a new version will conflict with an X lock on n, but will not conflict with locks on any of the existing n_is. The exact mechanism of the conflict is based on intention mode locks, as presented in [GRAY76, KORT83].
- Write specific version: This mode of access is represented by W. Concurrent reading and writing is, of course, not allowed. W mode does, however, allow concurrent reading of a dirty snapshot. If the writer needs to disallow such accesses, it may use X mode on the node corresponding to the specific version.
- Write latest: This mode of access is the same as "write specific version", where the specific version is the latest version.
- Read specific version: This mode of access is represented by R.
- Read latest: This mode is implemented using "read specific version" in a manner similar to the implementation of "write latest".
- Read latest available: This mode uses R mode. The system needs to determine the latest version on which no locks other than R and D locks are held. This information is available in the lock manager's state information.
- Read dirty snapshot: This mode of access is represented by D mode. A dirty snapshot is a copy of the datum that may contain uncommitted changes by concurrent transactions.

Note that the presence of D mode implies that we allow for non-serializable schedules even if two-phase locking is used. In what follows, we shall pay little attention to D mode under the assumption that it will be used only by read-only transactions. Thus, transactions that use D mode will not generate an inconsistent database.

Suppose that we support the granularities shown in Figure 5. Consider the case of a transaction checking a design object out of the design database. The checkout operation amounts to a read (R), on one of the versions. The version read is copied into the transaction's private database. The transaction has write access to the newly created private version. Eventually, the transaction will check in the object to the design database, thereby creating a new version. Concurrent checkout requests are allowed since they correspond to concurrent R lock requests.

Note that the approach to checkout and checkin presented above allows for the generation of non-serializable schedules, since two checkin operations on the same datum do not conflict. When we require serializability, we need to regulate further legal operations on versions. Among the rules we can add to the protocol for version creation are the following:

- Before a transaction may create a new version of a datum, it must hold the latest version of that datum in W mode.
- Before a transaction may create a new version of a datum, it must hold an X mode lock on the node representing all versions of the datum.

4. Lock Modes

In this section, we define the set of lock modes that we use in our concurrency control scheme. These modes are based on [KORT83], and Sections 4.2 and 4.3 are based heavily on that paper. After defining our set of locks, we propose new protocols based on the notion of *predicate-wise two-phase locking* that preserve database consistency as defined in [BANC85].

4.1. Correctness of a Compatibility Function

In the previous section, defined a lock compatibility function for the modes X, W, R, and D. We gave an intuitive argument justifying our choice of compatibility function. We now state precisely what we require of lock compatibility functions. Each lock mode that we have defined (X, W, R, D) corresponds to a database operation. For this reason, we call them *operational* modes. The operational lock modes define the amount of semantic information available for concurrency control.

Definition: A lock compatibility function COMPAT is *correct* if all schedules for all sets of two-phase transactions are serializable, provided that compatibility is observed.

Because of the inclusion of D mode, the compatibility function of Figure 7 is *not* correct. However, if we consider the restriction of COMPAT to X, W, R, then the function is correct.

Although we are not, in general, interested in serializability for CAD transactions, we shall see that the notion of correctness remains useful in defining the class of "acceptable" lock compatibility functions.

4.2. Definition of Intention Lock Modes

We define intention mode locks in order to ensure that locks taken at different granularities do not conflict. For example, we must avoid a situation in which a transaction is allowed an X mode lock on a relation while another transaction holds a W mode lock on one record of the relation. We define one intention mode for each operational mode. We then construct a lock compatibility matrix for this extended set of modes.

Let INITIAL denote our initial set of lock modes. For each mode a in INITIAL, we define a new mode called *intend-a* mode, denoted I_a. Let INTENT denote the union of INITIAL and the set of intention modes constructed from INITIAL. Let COMPAT be the initial lock compatibility function mapping INITIAL x INITIAL to {*true*, *false*}. The first argument to COMPAT signifies the mode of lock being *requested*. The second argument signifies the mode of lock *already held* on the data item in question. We extend COMPAT to INTENT as follows: Let p and q be two lock modes. Then COMPAT(p,q) is

- *true* if both p and q are intention modes, that is, if there is a mode P and Q in INITIAL such that $p = I_P$ and $q = I_Q$.
- COMPAT(P,q) if q is in INITIAL and there is a mode P in INITIAL such that $p = I_P$.
- COMPAT(p,Q) if p is in INITIAL and there is a mode Q in INITIAL such that $q = I_Q$.

4.3. Protocols and Intention Modes

To illustrate the intuition behind the use of intention modes, consider the special case in which the granularity DAG happens to be a tree. We require all transactions to observe the following rule, called the *tree/parent rule*.

Definition: A transaction observes the *tree/parent rule* if it does not request a lock on a node unless it already holds a lock on the parent of the node in the corresponding intention mode.

This rule does not apply to the root of the tree (since the root has no parent).

Thus, if transaction t is to be allowed to lock a node n in mode a, t must already hold an I_a lock on the parent of n. The tree/parent rule implies that transactions must begin locking at the root and take intention locks along tree paths to those nodes that it needs to lock in one of the operational modes (X, W, R, D).

The above rule is not sufficient to ensure a correct concurrency scheme. Suppose transaction t_1 follows the tree/parent rule to lock relation rel-1 in X mode and then releases all of its I_X mode locks. Another transaction t_2 could take an X mode lock on the root, thus locking the entire database implicitly. Therefore, we impose the *leaf-to-root node release rule* [GRAY76, KORT82]:

Definition: A transaction t observes the *leaf-to-root node release rule* in a DAG G if:

- for each node n of G, t is two-phase with respect to n. (i.e., no node is locked, unlocked, and re-locked).
- for each node n of G, t does not unlock n while it holds a lock on a child of n in G.

These definitions allow us to characterize the correctness of a lock compatibility function as follows:

Definition: A lock compatibility function is *tree-correct* if all schedules for all sets of transactions that

- are two-phase
- follow the tree/parent rule
- follow the leaf-to-root node release rule

are serializable provided that compatibility is observed. when we do not require serializability.

Theorem: The lock compatibility function for INTENT (X, W, R) is tree correct.

Proof: This is a corollary to a theorem in [KORT83].

In order to extend this locking scheme to DAGs, we need to replace the tree/parent rule. Since a DAG node may have many parents, we need to define the number of parents that must be locked. The most general form of the requirement that must be satisfied is that for any two conflicting INITIAL modes a and b, the number of parents that must be locked in I_a mode plus the number of parents that must be locked in I_b mode must be greater than the number of parents. For our purposes, we shall use only the following rule. We require intention mode locks on only one parent for R and D, but we require locks on all parents for W and for X. Furthermore, we restrict implicit locking. In order for a node to be locked implicitly, in R or D mode, only one parent of the node need be locked. However, all parents are required for implicit locking in W or X mode. We call our DAG analog of the tree/parent rule the *biased-parent rule* since we are "biased" in favor of readers. We choose to favor readers in this way because reading is a more frequent activity than writing. Thus, we anticipate that providing fewer requirements for reading will result in improved system performance.

Definition: A transaction observes the *biased-parent rule* if it does not request a lock on a node n in modes R, I_R, D, or I_D unless it already holds a lock on a parent of the node in the corresponding intention mode and it does not request a lock on a node n in modes X, I_X, W, or I_W unless it already holds a lock on all parents on the node in the corresponding intention mode.

The above rule leads to a notion of DAG-correctness:

Definition: A lock compatibility function is *DAG-correct* if all schedules for all sets of transactions that

- are two-phase
- follow the biased-parent rule
- follow the leaf-to-root node release rule

are serializable provided that compatibility is observed.

The following theorem follows directly from [KORT83]:

Theorem: INTENT (X, W, R) is DAG-correct.

4.4. Predicatewise Two-Phase Locking

We relax our requirement of *serializability* by replacing it with a requirement of *preservation of the consistency constraint*. In the model of [BANC85], the definition of each transaction includes an invariant and a partial order on the steps and sub-transactions of the transaction. We require that a transaction preserve its invariant if it is run alone. Initially, we shall assume that transactions are not nested. We consider nested transactions in Section 6.

Although we refer to the invariant as *the* consistency constraint, many practical invariants are a conjunction of relatively simple consistency constraints. This motivates us to put each invariant C into *conjunctive normal form*, that is, we write C as a conjunction of predicates $c_1, c_2, \ldots, c_n$, such that the c_i do not contain any "ands". It is an elementary fact of mathematical logic that we can put any C into this form. We refer to each c_i as a *conjunct*.

Before we introduce our main protocol, we present a special case in which each conjunct is expressed in terms of an individual object. We do not claim that this is frequently true in practice; rather we wish to illustrate the technique we shall use in the general protocol. Consider the history of accesses to one particular object by a set of transactions. If we can show that this history is equivalent to one created by a serial execution of the transactions, then we know that our consistency constraint is preserved. Note that this does *not* imply serializability since the equivalent serial ordering may be different for different objects. In this case, we can consider the following protocol:

- Two-phase locking with respect to versions: A transaction is required not to request any lock on any node pertaining to a multiversion granule after it has released a lock on that granule.

Note that this is a very weak two-phase requirement. That is, this requirement imposes fewer limitations on the legality of schedules than does standard two-phase locking. It is applied to each multiversion granule *individually.*

Although the above rule is not very general, it suggests a fruitful approach to concurrency control without global serializability. We define localized sections of the database on which two-phase locking is required. To the extent that these sections of the database are small, we have gained in potential concurrency over standard two-phase locking.

We now extend our approach to arbitrary predicates.

- Two-phase locking with respect to sets of predicates (Predicatewise 2PL): For each conjunct, c_i, let d_i denote the set of data items mentioned in c_i. This protocol requires that two-phase locking be observed with respect to each d_i set of which a member is accessed by the transaction.

Note that this is weaker than standard 2PL, but stronger than two-phase locking with respect to versions.

We now introduce a new notion of correctness and a new multiple-granularity locking protocol based upon predicatewise 2PL, the leaf-to-root node release rule, and the biased-parent rule. We begin by defining a notion of correctness for a single-granularity locking scheme.

Definition: A lock compatibility function is *predicatewise correct* with respect to a consistency constraint C if all schedules for all sets of transactions that:

- preserve C
- are predicatewise two-phase

preserve C provided that compatibility is observed.

Now, we extend the above notion to a multiple-granularity locking scheme represented by a DAG:

Definition: A lock compatibility function is *predicatewise-DAG correct* with respect to a consistency constraint C if all schedules for all sets of transactions that:

- preserve C,
- are predicatewise two-phase,
- follow the biased-parent rule, and
- follow the leaf-to-root node release rule

preserve C provided that compatibility is observed.

When we use the term *predicatewise correct* without reference to a specific constraint C, we mean predicatewise correct with respect to all possible constraints C. We are now able to justify our original choice of a lock compatibility function:

Theorem: The compatibility function of Figure 5 for (X, W, R) is a predicatewise correct compatibility function.

Proof: Let s be a schedule. Let C be the consistency constraint and let c_i $(i=1,...,n)$ denote the conjuncts in a conjunctive normal form representation of C. For each set d_i of data items referenced by conjunct c_i, let s_i denote a schedule formed from s as follows: Take those steps from s that access a data item in d_i and list those steps in the same order in which they appear in s. Note that a particular step in s may appear in several of the s_is. Since all transactions in the set T that generated s are predicatewise two-phase, they are two-phase with respect to d_i. For each transaction t in T, let t_i denote a transaction formed by taking only those steps of t that appear in s_i. Since, t_i accesses only data items in d_i, and t_i is two-phase with respect to d_i, t_i is a two-phase transaction. Therefore, s_i is a schedule for a set of two-phase transactions. Since we know that the compatibility function is correct, and is observed in s_i, s_i must be serializable. Each s_i thus preserves c_i. Furthermore, since the steps in s_i include exactly the steps in s that access a data item in d_i, it follows that s must preserve c_i as well. Since i was chosen arbitrarily, s preserves c_i for all i and therefore, s preserves the conjunction of the c_is, which is C.

We generalize the proof of the above theorem to show the following more general result.

Theorem: If a lock compatibility function is correct then it is predicatewise correct.

Proof: Let C be a consistency constraint and let c_i $(i=1,...,n)$ denote the conjuncts in a conjunctive normal form representation of C. Let d_i denote the set of data items referenced by conjunct c_i. Assume that COMPAT is not predicatewise correct with respect to C. Then there is a schedule s for a set T of predicatewise two-phase transactions such that s fails to preserve some c_i. Each t in T must preserve c_i if there is no concurrency since C is the consistency constraint. Define a set R of transactions as follows: For each t in T, create a transaction r consisting of those steps of t that access data in d_i. R is the set of all such transactions r. Define a schedule p for R by deleting from s those steps that do not pertain to data in d_i. Since s does not preserve c_i, neither does p. However, all transactions in R are two-phase since all transactions in T are two-phase with respect to d_i. But then p is a counterexample to the assertion that COMPAT is correct.

The above theorem implies that we can use any correct compatibility function for predicatewise two-phase locking, regardless of the consistency constraint. However, since in a CAD database, we are interested in multiple granularities of locking, we must consider the extension of a predicatewise correct compatibility function to a predicatewise DAG-correct compatibility function. In [KORT83], it was shown that the extension of correct compatibility functions to INTENT results in a DAG-correct compatibility function. The following theorem is an analogous result for predicatewise correctness.

Theorem: Let INITIAL be a set of basic lock modes and let COMPAT be a given lock compatibility function for INITIAL. If COMPAT is predicatewise correct, then the extension of

COMPAT to INTENT, as defined above, is predicatewise-DAG correct.

Proof: Let s be a schedule for a set T of two-phase biased-parent rule observing transactions. Construct a schedule p from s as follows: Delete all steps involving the request or release of intention mode locks. Replace each step that locks a granule of data in an operational mode with a series of steps that locks explicitly all granules (at the finest granularity) locked implicitly by the step being replaced. Schedule p is equivalent to s since there is no change to steps that result in modification to the database. Furthermore, if the compatibility function INTENT was observed in s, then COMPAT must be observed in p. Since COMPAT is predicatewise correct, p must preserve the consistency constraint and thus, so must s.

We note without proof that the above theorems still hold if we used an unbiased parent rule rather than a biased parent rule as the basis of our definition of predicatewise correctness. This follows from a theorem of [KORT83].

5. Conversions and Deadlock

It is often the case that a transaction will read a datum, do some computation and some other database accesses, and then write that datum. If we require that a write lock be used for this purpose, we reduce potential parallelism. Yet, if we allow conversions from read mode to write mode, we may introduce deadlocks involving updaters. A simple example of this is two transactions that obtain read locks on a datum and both wish to convert the read lock to a write lock. [GRAY81a] reports on experiments that show that a large percentage (97 percent) of real-world deadlocks may result from such conversions. The class of *update-mode* locks [KORT83] is designed to eliminate most (though, unfortunately, not all) deadlocks resulting from conversions, while minimizing the impact that this has on the amount of parallelism. Given a set of lock modes (such as INITIAL or INTENT) we generate an update mode for every pair of lock modes in our given set. If a and b are given lock modes, then U_a^b is a mode which allows exactly the privileges of a mode but indicates that the transaction plans to convert this lock to b mode at some point in the future.

As an example, consider U_R^W mode (called *update* mode). This mode allows its holder the privileges of R mode, that is, the right to read the locked datum. This mode is not compatible with itself, thereby avoiding the simple deadlock scenario we noted above. However, update mode is designed to allow the update transaction to read the datum concurrent with transactions holds a R mode lock. [KORT83] defines COMPAT(U_R^W,R) to be true, but COMPAT(R,U_R^W) to be false. This prevents a series of readers from delaying the updater indefinitely. In the experiments of [GRAY81], approximately 76 percent of the observed deadlocks could have been avoided by the use of update modes.

The notion of update mode that we have just defined appears to be the appropriate form of update mode for short-duration transactions. We conjecture that for long-duration updaters, it makes sense for COMPAT(R,U_R^W) to be true. Consider, for example, a case in which a long-duration transaction has checked out an object and will be overwriting the object upon checkin of the object (i.e., it will not be creating a new version). In such a case, we wish to allow short duration transaction to have read access to the object without waiting for the object to be checked back in. If the long-duration transaction uses update mode, read access may proceed concurrently with the long-duration transaction until the point that the long-duration transaction upgrades its lock via a lock conversion in order to be able to check in its updated version. However, transactions that need to modify the datum locked in update mode will be forced to wait.

We now give a general definition of an extension of a compatibility function COMPAT to the update modes. Let UPDATE denote the set of lock modes consisting of our given set of lock modes and all update modes U_a^b for which a is a weaker lock mode that b. Then, using the definition of [KORT83], COMPAT(U_a^b,U_c^d) = COMPAT(a,d).

Theorem: Let MODES be a set of lock modes with a predicatewise-DAG-correct compatibility function COMPAT. Then the extension of COMPAT to UPDATE is also predicatewise-DAG-correct.

Proof (sketch): The proof follows directly from the observation that for all mode a and b, U_a^b is a mode that allows the same accesses as a but is strictly more restrictive that a. Thus any counterexample to the predicatewise DAG correctness of UPDATE would also be a counterexample to the predicatewise DAG correctness of COMPAT.

6. Implementation of the Protocol with Nested Transactions

The above discussion of lock modes and protocols allows us to define a correct approach to the generation of non-serializable schedules. However, a naive implementation of a lock manager might result in a subtransaction waiting for its parent, thereby creating unnecessary waits and deadlocks. For example, if the lock manager treats the parent and its child as two distinct, unrelated transactions, the child would not be able to execute against data held by the parent.

We want to allow subtransactions to access data concurrently with parent transactions, subject, of course, to authorization by the parent. In other words, it may be legal for two distinct transactions to hold locks on a datum in what appear to be incompatible modes, provided that one transaction is a subtransaction of the other. In this section, we define those cases in which such concurrent access is to be permitted and propose a scheme to implement it.

In the CAD transaction model of [BANC85], a system has a collection of *project* transactions that operate with a large degree of isolation from each other. Conflicting lock requests from distinct project transactions are treated as conflicting requests. A project transaction locks a datum, not to access it directly, but to allow its subtransactions to access the datum. In other words, a lock taken by a project transaction is intended to restrict access to the datum by other project transactions. Since a project transaction has many subtransactions executing concurrently, it is necessary to impose locking at the subtransaction level. Although a lock taken by one subtransaction must conflict with an incompatible lock requested by another subtransaction, it must not conflict with the lock held by the parent project transaction.

We implement locking in our transaction model by defining a nested form of the protocol we presented above. Each transaction or subtransaction is allowed access to a subset of the entire database defined by a sub-DAG of the DAG for the entire database. Conceptually, there is a separate lock manager for each level of nesting. A lock request is *legal* if all parents of the requesting transaction already hold the lock being requested. A lock request is granted if the lock manager for the appropriate nesting level allows it. In other words, a legal lock request is checked only against requests made by other transactions at the same level of nesting within the same parent (that is, against sibling transactions).

In practice, we do not want to implement multiple lock managers due to the excessive overhead involved. Instead, we use a hierarchical transaction naming scheme that allows a single lock manager to simulate the various lock managers required by our scheme. Let t_1, t_2, ..., t_n denote the set of project transactions. Each subtransaction of a project transaction (a *cooperating* transaction in the model of [BANC85]) is named uniquely within its project. The globally-unique name of the transaction is the concatenation of its parent's name with its own locally-unique name. Thus, the cooperating transactions within project transaction t_1 are named $t_{1,1}$, $t_{1,2}$, ..., $t_{1,m}$. In general, the name of a transaction includes the full name of its parent as a prefix. Using this naming structure, it is a simple matter for a single lock manager to perform the following functions:

- check legality of requests: If the lock manager is designed to accept only legal requests, then to verify the legality of a new request it suffices to see that the parent of the requesting transaction has the appropriate lock.
- check legality of lock release requests: If a transaction attempts to release a lock, the lock manager can check to see if a subtransaction of the requester still holds a lock on the datum. If so, the lock manager should disallow the request.
- simulate a set of nested lock managers: A legal lock request is checked for compatibility in the same way as a standard lock manager would. However, if the request turns out not to be compatible with one or more currently-held locks, the name of the transaction holding

the lock is compared with the name of the transaction requesting the lock. If the name of the transaction holding the lock is a prefix of the name of the transaction requesting the lock, then the incompatibility is ignored.

It is important to note that the above technique is based on the assumption that all lock managers being simulated follow the same set of protocol rules.

Our CAD transaction model allows for several kinds of subtransaction. Variations in the nature of concurrency control required are accommodated easily in our concurrency control scheme if the semantics of the lock modes remain the same and the variation is only in the presense or absence of restrictions on how long a lock may be held (e.g. degrees of consistency in [GRAY76]).

A more serious problem is presented by certain forms of the client/subcontractor relationship of [BANC85]. There may be a partial (or total) ordering among the subcontractors of a client (parent) transaction. In order to preserve this ordering, [BANC85] proposes the use of a virtual timestamping scheme. Timestamps are assigned to each client/subcontractor transaction based on the partial order. By use of the standard timestamp-ordering protocols, it is then possible to enfore the partial order.

It is necessary to integrate these two concurrency control schemes. Because of the absence of waits in timestamp schemes, it is never necessary to involve the timestamp scheme in the deadlock detection algorithm of the lock manager. However, each concurrency manager must inform the other of any transaction aborts that it initiates. Within the timestamp scheme, a transaction must not be given access to data that is not locked by its closest ancestor running under a locking protocol. Similarly, a transaction running under locking must not be given a lock that its closest ancestor running under timestamping could not access. These two conditions are easy to test via simple queries of the concurrency control's internal data structure. Testing these conditions corresponds directly to tests that have to be performed in standards applications of each of these schemes.

7. Implications of Concurrency Control for Recovery

In traditional database systems, crash recovery requires that all transactions active at the time of a crash be aborted. This presents a severe problem in systems that include long-duration transactions. The amount of work that is lost in aborting a long-duration transaction is significant. Further, in a CAD environment, long-duration transactions include not only work done by the system, but also work done by designers. Thus, abortion of long-duration transactions is undesirable not only from a performance standpoint, but from a human-factors standpoint.

In order to minimize the amount of work that is lost due to a crash, the notion of a *save point* is proposed in [GRAY78, GRAY79]. Under the save-point scheme, a transaction may request that the system save the internal state of the transaction. In the event of a crash, the transaction is restarted from the most recent save point possible.

In general, the implementation of save points is complex. Not only must the internal state of the transaction be saved, but also all subsystems of the database system must save all data pertaining to the transaction. For example, the lock manager must save all locks taken by the transaction requesting a save point. At recovery time, it may not be possible to restart the transaction from the most recent save point prior to the crash. Let us consider transaction t_1, which has read data written by transaction t_2. If t_2 has not yet committed, and cannot be restarted at a point beyond the writing of the data read by t_1, then it is not possible to restart t_1 at any point *after* the reading of the data written by t_2.

There are several approaches to dealing with this problem. One is to construct a dependency graph which is used at recovery time to choose the correct save point at which to restart each transaction. Alternatively, we can impose a locking scheme which requires that all lock releases be delayed until the end of the transaction. In this case, it is always possible to restart a transaction from its most recent save point.

In the CAD transaction model of [BANC85], long-duration transactions are composed (at the lowest level of nesting) of a set of short-duration transactions. This fact allows us to solve the transaction-restart problem more simply. If a long-duration transaction is active at the time of a crash, any short-duration subtransactions of the long transaction are aborted. The results of all committed subtransactions are restored (by means of a redo operation, if necessary). Thus, in effect, the termination of a short-duration transaction represents a save point for the long-duration transaction that contains it.

The state of a long-duration transaction includes a record of which subtransactions have been completed. This information is obtained from the log at recovery time by checking for commit records for subtransactions [GRAY81b]. This data represents most of the long-duration transaction's state, so the overhead of implementing a save point at the termination of each subtransaction is minimal. However, in order to be able to restart the database system, it is necessary to restore the lock manager's record of locks held by long-duration transactions. Locks held by short-duration transactions need not be restored, since those transactions will be aborted. Therefore, we treat locks taken or released on behalf of a long-duration transaction as we treat database accesses. Specifically, each lock and unlock request by a long transaction is written to the log. These log records are forced to disk in accordance with the write-ahead log protocol [GRAY81c].

A second problem pertaining to recovery in systems with long-duration transactions is management of the log. In a traditional database system, it is possible to perform a *checkpoint* of the database and purge all log records that were written before the checkpoint. In case of a catastrophic failure, recovery is accomplished by reloading the database from the checkpoint tape, undoing uncommitted transactions, and redoing committed transactions using the log. When all transactions are short, relatively few log records need to be kept for each transaction active immediately following a checkpoint. These records are all near the end of the log. Thus, it is reasonable to treat the log as a sequential-access file.

In a system with long-duration transactions, modeling the log as a sequential file has two disadvantages:

- It may force an unnecessarily large number of log records to be kept on-line after a checkpoint.
- An undo operation occurring due to the abort of a long-duration transaction (rather than due to recovery from a crash) must scan a large number of irrelevant log records.

Thus, the log must be treated as a random-access file indexed by transaction identifier. Although this complicates checkpointing and logging to some degree, all of the code required to implement this is already available within the database system to support the database itself.

8. User Interface

The approach we have presented for concurrency control benefits from having a large amount of information about transactions. The more information available regarding the possible future accesses of transactions, the higher the degree of concurrent execution possible. Below we list the kinds of information that are useful:

- Transaction duration: It is useful to know if a transaction is of short or long duration. We may assume that all designer's transactions are long durations transactions and that each command entered by the designer as part of this long-duration transaction is a short duration subtransaction. In those cases where the designer begins a long-duration transaction it is likely that he will either open a new window (thereby declaring a new long transaction) or execute a single command that invokes a long transaction that was coded in advance by an application programmer. The application programmer can place a declaration of expected transaction duration in his code. (The idea here is that the designer never gives an explicit transaction duration declaration, but we are willing to ask application programmers to do this. Such declarations need not be complex -- a single statement in the application programmer interface that translates to a system call should suffice.)

- Completion of access to a datum: If a transaction access a datum and completes all work with that datum prior to the end of transaction, this information is useful. It allows early release of locks on that datum if such releases are consistent with the two-phase locking requirements imposed by PW2PL. A designer-level command of "release d", where d is a data item, can be included. Most designers will not bother with this command since it does not help them and is only a public service. However, such commands may be useful if the designer receives a phone call from a colleague asking if he is done with datum d. The release command is the means by which the designer may respond "yes" to such a query.
- Negative information: It is as useful for the system to know what a transaction will NOT access as it is to know what a transaction will access. Negative information, along with the release command, allows for early lock release under PW2PL. Calls for the declaration of such information can be placed in the programmer interface. Furthermore, the compiler may deduce such information via data-flow analysis and the compiler itself may insert declarative system calls.

9. Summary

We have explored concurrency control schemes that admit the special requirements of CAD objects and that exploit our model of transactions. We presented an extension to the theory of multiple-granularity locking, which incorporates CAD objects as lock granules. Then we explored some techniques for increasing the degree of parallelism in concurrently executing transactions, when we remove serializability as the criterion for database consistency.

We defined predicatewise two-phase locking in order to allow the definition of protocols that ensure the preservation of consistency constraints while allowing greater parallelism than under traditional approaches to concurrency.

References

[BANC85] Bancilhon, F., W. Kim, and H. F. Korth, "A Model of CAD Transactions," *Proc. 11th International Conference on Very Large Data Bases*, pp. 25-33 (1985).

[BERN81] Bernstein, P.A., and N. Goodman "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, **13:2** (June 1981), pp. 185-221.

[GRAY76] Gray, J.N., R.A. Lorie, G.R. Putzolu, I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", in Nijssen, G.M., ed. *Modeling in Data Base Management Systems,* pp. 365-394, also RJ1606, IBM Research Laboratory, San Jose, CA.

[GRAY78] Gray, J.N. "Notes on Data Base Operating Systems," RJ2188, IBM Research Laboratory, San Jose, CA., February 1978.

[GRAY79] Gray, J.N., P. McJones, M. Blasgen, R. Lorie, T. Price, G.R.. Putzolu, and I.L. Traiger, "The Recovery Manager of a Data Management System", RJ2623, IBM Research Laboratory, San Jose, CA.

[GRAY81a] Gray, J.N., P. Homan, H. F. Korth and R. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock," Oral presentation, *5th Berkeley Workshop on Distributed Databases and Computer Networks*, also, RJ3066 IBM Research Laboratory, San Jose, CA.

[GRAY81b] Gray, J.N., "The Transaction Concept: Virtues and Limitations," *Proc. 7th International Conference on Very Large Data Bases*, pp. 144-154 (1981).

[GRAY81c] Gray, J.N., et al., "Recovery Manager of a Data Management System", *ACM Computing Surveys,* **13:2,** pp. 223-242.

[HASK82] Haskin, R. and R.A. Lorie, "On Extending the Functions of a Relational Database System," *Proc. ACM SIGMOD International Conference on Management of Data*, June 1982, pp. 207-212.

[KATZ84] Katz, R. and S. Weiss, "Transaction Management for Design Databases," working paper, 1984.

[KIM84] Kim, W., R.A. Lorie, D. McNabb, and W. Plouffe, "A Transaction Mechanism for Engineering Design Databases," in *Proc. 9th International Conference on Very Large Data Bases*, August 1984.

[KORT82] Korth, H.F., "Deadlock Freedom Using Edge Locks", *ACM Transactions on Database Systems,* **7:4,** (Dec 1982), pp. 632-652.

[KORT83] Korth, H. F., "Locking Primitives in a Database System," *Journal of the ACM,* **30:1** (Jan 1983), pp. 55-79.

[MOSS81] Moss, J.E.B., "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, MIT, April 1981.

[PAPA82] Papadimitriou, C., and P. Kannelakis, "On Concurrency Control by Multiple Versions," *Proc. ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems*, pp. 76-82.