A Study of the Concurrency Control and Recovery Algorithms in Nested Transaction Environment

SANJAY KUMAR MADRIA

School of Computer Science, University Sains Malaysia, 11800 Minden, Penang, Malaysia
Email: skm@cs.usm.my

In this paper, we present a study on the concurrency control and recovery algorithms in nested transaction environment. We have reviewed the work done in the area of nested transaction modelling, its applications in object-oriented and mobile databases, and in workflow models. We have contrasted various nested transaction models by discussing their advantages and disadvantages. We have outlined some important future research directions in the area of nested transaction processing.

Received July 1, 1997; revised December 2, 1997

1. INTRODUCTION
Transaction processing, concurrency control and recovery issues have played a major role in conventional databases [1], and hence have been an important area of research for many decades. However, with the increasing use of advanced database applications such as CAD/CAM, large software design projects, object-oriented databases both in centralized and distributed environments, there is a vital need for better algorithms for handling the new applications more efficiently. Nested transactions [2] have been shown to play an important role in such applications. For example, in object-oriented databases, objects are active and have a nested structure, thus nested execution is an important concern for transaction management in such systems. Similarly, in long transaction environments such as CAD/CAM [3, 4] and software development processes [5], nested transactions can play an important role in improving concurrency and in providing efficient recovery which is not possible with single level transactions. Nested transactions have been used in mobile computing [6], and recently in workflow models [7].

The rest of the paper is organized as follows. In Section 2, we discuss nested transaction models. In Section 3, we review nested transaction concurrency control algorithms and their correctness. Section 4 discusses the nested transaction applications in object-oriented and mobile databases and in workflow models. In Section 5, we discuss nested transaction recovery algorithms. Section 6 presents some future research directions. We present conclusions in Section 7.

2. NESTED TRANSACTION MODELS
There has been an increase in interest in the study of the possibility of giving more structure to transactions and redefining the basic unit of atomicity. The theory of nested transactions [8] allows the benefits of atomicity to be available within a transaction. Atomic objects can be created, and operations on these objects are guaranteed to be serializable even though they execute concurrently. These types of subtransactions were first suggested by Davies [9] under the heading of ‘spheres of control’. Nested transactions have been implemented in system R [10], Argus [11, 12], Camelot [13], Clouds [14], Locus [15] and Eden [16] etc., and are widely accepted as a suitable mechanism for reliable distributed transaction processing systems. Recently, a commercial system using nested transactions was released called Encina [17], a TP-monitor from TransArc where most of the ideas of Camelot have been implemented.

In a nested transaction model [2]:

- Each nested transaction consists of primitive actions (e.g. read and write) or subtransactions that are also nested transactions. A transaction may contain operations to be performed concurrently, or operations that may be aborted independently of their invoking transaction. Such operations are considered as subtransactions of the original transaction. A transaction and its subtransactions have the normal parent–child relationship. The tree defined by this parent–child relation is called a transaction tree in which nodes represent transactions and the edges represent the nesting relationship amongst the transactions. In referring to this tree, standard terminology such as ‘leaf node’, ‘internal node’, ‘ancestor’ (superior) and ‘descendant’ (inferior) are used. The notion of a least common ancestor (l.c.a.) of two nodes is also used. In a transaction tree, leaves can occur at different levels. A top level transaction
might itself be a leaf representing a single data access or might invoke both a subtransaction and a data access. Failure of subtransactions may result in invocation of alternative subtransactions that could replace the failed ones to accomplish the successful completion of the whole transaction. Such types of transactions are termed nested transactions [2]. The atomic property of a transaction ensures consistency in a database system in the presence of failures and concurrent operations.

- Each set of sibling transactions or subtransactions is supposed to execute in a serializable fashion. Transactions can terminate either normally by committing or abnormally by aborting. A transaction is not allowed to commit until all of its children have been terminated. Each set of sibling transactions runs as if all the transactions that committed ran in serial order and all the transactions that aborted did not run at all. An external consistency property is also required for each set of siblings, ensuring that if a transaction waits for a child T to complete before invoking another child T', then T is before T' in the apparent serial order.

- ACID-properties [18] are fulfilled for top level transactions and a subset of them are defined for subtransactions. A subtransaction appears atomic to the other surrounding transactions and may commit and abort independently. Aborting a subtransaction does not affect the outcome of transactions not belonging to the subtransaction’s hierarchy. The durability of the effects of a committed subtransaction depends on the outcome of its superiors. Even if it commits, aborting one of its superiors will undo its effects. A subtransaction’s effect becomes permanent only if it commits and the enclosing top level transaction completes.

- When a transaction is aborted, its computation is terminated and its parent notified. However, the parent continues its execution. That is, a child transaction can abort without forcing its parent to abort. In fact, the parent transaction can take different steps in comparison to when the child commits. The parent may choose to restart the subtransaction. Nested transactions also allow localized handling of transaction aborts. This allows for possibilities such as attempting a piece of computation on one computer and redoing only that piece if the computer fails.

- The highest level user-defined transactions are modelled as children of the root, T. The root T has a special role. T models the environment of the nested transaction system (the ‘external world’) from which requests for the invocations of top level transactions originate and to which the results of these transactions are sent. Since it has no parent, T may neither commit nor abort. The classical transactions (without nesting) of concurrency control theory appear in this model as children of T. It is natural to reason about T in the same way as about all the other transactions.

Some of the main advantages of using nested transactions are as follows:

- Using nested transactions, the aborts of subtransactions can be handled independently and locally. This must be considered in contrast to single level transactions where abortion implies losing everything that was done in the whole transaction unless some special mechanism is adopted. This aspect of nested transactions is also called inter-transaction recovery [19].

- Nested transactions provide a mechanism to distribute activities in distributed databases at different sites.

- They provide synchronization between concurrently running parts of the same nested transactions. This is known as intra-transaction concurrency.

- In object-oriented databases, if the execution of methods is seen as a transaction then the execution of methods generates a hierarchy that corresponds to nested transactions.

- A nested transaction gives great flexibility in terms of safe decomposition of application programs whose modules may be designed and implemented independently. One important aspect of this modularity is that one can associate semantics with each method or subtransaction.

Nesting in transactions corresponds either to the nesting of procedures or to the nesting of layers of data abstractions.

- In the first kind [2, 11, 13], a subtransaction’s updates are not visible outside its parent, and therefore availability is restricted. If the parent aborts, the subtransaction is also aborted. That is, abort operations are handled by aborting the transactions of all underlying operations.

- In the second kind [20, 21], a subtransaction’s modifications are visible to other transactions at the same level of data abstraction as soon as it commits, even if its parent is still active. Hence, it provides more availability in comparison to the first model. However, later if the parent aborts, a suitable compensatory operation is initiated to remove the effects of already committed subtransactions. Since basic (i.e. read and writes) locks are released early and have possibly been acquired by other transactions, abort has to take place in the form of inverse or compensatory operation. In this case, an abort has to be possible without affecting any other transaction.

A more complex notion of nesting, emphasizing level of data abstraction, is used in system R [10] and has been studied in a number of papers that include work by Beeri et al. [20], Moss et al. [22] and Weikum [21].

- In System R, locking is applied twice. First, on tuples until EOT and then on pages for the scope of each tuple action. Hence tuple actions can be regarded as subtransactions. Page locks are released before the commitment of the entire application transactions. This technique of long tuple locks and short page locks has been called ‘open nested transaction’. The page level locks guarantee serializability of tuple actions, thus
ensuring that they can be considered to be elementary and allowing for tuple level concurrency. However, nesting is restricted to two levels only and the recovery scheme is complicated.

- In [20], a description of nested transactions and basic techniques for proving serializabilty of nested transactions was presented. The system is organized using multiple levels of abstraction, with concurrency control performed separately at each level. The nesting in such systems corresponds to levels of data abstraction and allows replacement of the entire subtree of nesting activity by a single action as well as the reordering of actions in a history. This makes the technique more complicated. Object-oriented database systems like ORION [23] and O2 [24] ensure serializable executions and use a locking protocol based on multigranularity locking. The model in [20] does not include recovery in the work presented.

- Weikum presented a multi-level transaction model [21, 25] to exploit layer specific semantics at each level of operation nesting. The model takes into account the commutative properties of the semantics of operations at each level of data abstraction to achieve a higher degree of concurrency. If two operations at the same higher level commute then their conflicting descendants at the same lower level will be allowed to execute since they will not introduce any inconsistencies. Hence, this layered transaction model achieves higher concurrency. Formally, a multi-level transaction is defined as follows: it is a balanced tree of actions in which all nodes at the same depth correspond to operations of the same level of abstraction. Edges in a transaction tree represent the implementation of an operation by a sequence of operations at the lower level. In this model, a subtransaction is allowed to release locks on finishing before the commitment of higher level transactions. In case a higher level transaction aborts, the effect of the aborted transaction has to be undone by a compensatory transaction.

3. NESTED TRANSACTION CONCURRENCY CONTROL ALGORITHMS

3.1. Nested transactions concurrency control using locking

- In [2], Moss presented a concurrency control algorithm using two phase locking [26] for a nested transaction environment. In brief, the locking rules are as follows. A transaction can acquire a lock on the data object on request if no other transaction holds a conflicting lock on the data object. Also, all the retainers of the conflicting locks are ancestors of the requested transaction. A transaction can terminate only after all its children are terminated (committed or aborted). When a transaction commits, and if it is not a top-level transaction, its locks are inherited by the parent (so that it retains the inherited locks). If it aborts, its locks are discarded. If it is a top-level transaction, its locks are discarded on its termination.

- In [27, 28] Lynch has presented a complete proof of the exclusive locking algorithm for nested transactions. Moss [2] has also extended two-phase locking with separate read/write locks to handle nesting, which is the basis of data management in Argus [11]. A formal proof of Moss’s read/write algorithm for nested transactions appeared in [28].

- In [29], the read-update locking algorithm [30] has been generalized and a new commutative locking algorithm has been introduced to handle nested transactions. The paper defines a local atomicity condition for data objects, called dynamic atomicity. The atomicity theorem is used to show that if all the objects in the system are dynamic atomic then the system guarantees atomicity. Dynamic atomicity provides modularity as it allows one to verify the implementations of individual objects independently. It also allows the use of different implementation techniques and different algorithms in different objects as long as all the objects are dynamic atomic.

- The multi-granularity algorithm given in [31] has been extended to nested transaction systems in [32]. The algorithm considered file and record level granularity. The correctness proof shows that there is a possibility mapping to the abstract algorithm for commutativity-based locking given in [29]. The paper shows that some objects can use multi-granularity locking while others use Moss’s two-phase locking and others use different user-defined dynamic atomic algorithms. Some more related work by the same authors on concurrency control using predicate locks appears in [33, 34].

- Fekete et al. [35] have presented a serialization graph construction for nested transactions based on an input/output (I/O) automaton model [36]. The proof technique has the same form as in the classical theory; one must show that a graph having transactions for nodes and edges representing ordering between transactions is acyclic. The serialization graph model introduces the concept of visibility that hides the effects of a subtransaction to any other subtransaction until all its ancestors, up to the least common ancestor (l.c.a.), are committed. They have defined a new kind of serialization graph and proved that under certain assumptions, the absence of cycles in this graph is a sufficient condition to ensure the serial correctness of a system. They have applied their technique to verify the correctness of Moss’s read/write locking algorithm for nested transactions, and an undo logging algorithm that has not been previously proved for nested transaction systems.

Most of the above discussed algorithms appear collectively in [37].

- A formalization and correctness of a concurrency control algorithm for an open and safe nested transaction using an I/O automaton model has appeared in [38]. The nested transaction model uses a prewrite operation before an actual write operation to increase
the concurrency. The concurrency control is provided by extending Moss’s two-phase locking algorithm. The model also takes into account the buffer management operations which are controlled by the concurrency control algorithm. Non-access subtransactions, objects and the scheduler are modelled as I/O automata with the help of some pre- and post-conditions. The concurrency control algorithm has been proved serially correct in the sense of [8].

- In [39], nested transactions have been used in knowledge base systems (KBMSs). A concurrency control algorithm in a nested transaction environment using enhanced lock modes for KBMSs is presented. The enhanced lock modes capture the abstraction relationships’ semantics. The concurrency control method used here for nested transaction is the one proposed in [19] which allows for upward as well as controlled downward inheritance of locks. The downwards inheritance of locks makes objects manipulated by a parent transaction available to its children in contrast to Moss’s model [2] which allows only upward inheritance of locks. The model allows for an effective exploitation of intra-transaction concurrency inherent in a knowledge representation approach.

3.2. Nested transactions concurrency control using timestamps

Reed [40] has presented a multi-version timestamp concurrency control algorithm to provide nested transaction-based data management. Different versions of an object keep track of the ‘history’ of the object. A timestamp is associated with each version of an object and is used to allow concurrent execution of subtransactions. In [41], a formal analysis of the multi-version timestamp-based algorithm is given. The algorithm is proved to be correct by showing that the objects used in the algorithm are all static atomic. The atomicity theorem [8] is used to show that if all the objects in the system are static then the system guarantees atomicity. The proof techniques given are very general and can be applied to a large class of systems including those where different data objects are implemented independently, and where type of objects can be used to obtain increased concurrency.

3.3. Nested transactions and replication control algorithm

Gifford’s basic quorum consensus algorithm for data replication [42] is generalized by Goldman in [43] to accommodate nested transactions and transaction aborts. The presentation separates the treatment of replication entirely from concurrency control which helps to simplify the reasoning. The paper shows that any correct concurrency control mechanism may be used on the copies, considered as separate objects; the whole system will then appear to be atomic and non-replicated.

3.4. Nested transactions accessing data structures

- A concurrency control algorithm for accessing B-trees in the nested transaction model has been proposed in [44]. The notion of ‘strongly serially correct’ behaviour has been defined in [44] and has been used as the correctness criterion. The strongly correct behaviour says that a schedule \( \alpha \) is correct if there is a serial schedule \( \beta \) such that no transaction can tell the difference between \( \alpha \) and \( \beta \), and each data object, storing the value of a key, is updated by \( \alpha \) and \( \beta \) in exactly the same sequence. This definition requires the same schedule \( \beta \) for all the transactions \( T \) such that \( \alpha T = \beta T \). However, the work in [44] has some drawbacks. First, vertices and keys both are considered to be data objects but lock management on vertices is not implemented. Lock management on keys is said to be performed by the respective vertex manager but no lock seems to be acquired on the vertex when a key is not found. Each vertex and key needs to be locked to ensure that changing of data at a vertex is done in an atomic step to ensure serializability. Therefore, it is not clear how serializability is being guaranteed. Secondly, the compression process [45] which merges two adjacent vertices as a result of any deletion of keys is also not implemented.

- In [46], a nested transaction version of the concurrency control algorithm is presented using linear hash structures. The drawbacks of [44] have been removed. The new algorithm increases concurrency and also handles transaction aborts. In the algorithm, the locks have been considered at both key and vertex level. These locks have been implemented in a nested transaction environment using Moss’s two-phase locking algorithm and the locking protocols of the linear hash structure algorithm with lock-coupling technique. The algorithm has been implemented using an I/O automaton model. The linear hash structure algorithm in nested transaction environments is proved to be ‘serially correct’.

In all the algorithms discussed above, recovery from transaction aborts is considered but crash recovery is ignored.

4. NESTED TRANSACTIONS IN OBJECT-ORIENTED AND MOBILE DATABASES, AND IN WORKFLOW MODELS

In object-oriented databases, real life entities are represented by instances of abstract data types, called objects. An object consists of variables and methods or operations. Operations defined for these objects characterize the behaviour of the entities. The only interfaces to the objects are through these operations. The internal object structure is hidden from the users. Thus, complex real life entities can be modelled and manipulated easily without worrying about their implementation details. These methods can invoke atomic operations on the object variables or invoke other
methods. Thus, the execution of methods maps conveniently
into nested transactions. Therefore, nested executions are a
fundamental concern of transaction management on object-
oriented databases. Current applications are increasingly
using object-oriented databases to model non-standard
applications.

To capture the needs of emerging information intensive
applications such as CAD/CAM and object-oriented
databases etc., several extensions to the traditional data and
transaction models have been proposed. Abstract data type
models have been proposed to capture the semantics of
data in complex databases. Such data types have a rich
source of semantic information, and thus allow the design
of type-specific concurrency control schemes which enhance
concurrency within objects. These schemes exploit the
semantics information about the types and their operations.

- A generalization of the classical serializability theory
for handling object bases that exhibit a nesting structure
can be found in [47]. They modelled nested
execution of transactions. They defined a serialization
graph construction, and give an acyclic condition for
serializability. They have also presented a nested
two-phase locking for nested transactions which is
the generalization of [2]. In this protocol, locks are
requested for the atomic operations, which correspond
to the leaves in the nested transaction tree. When
a method execution terminates, its parent inherits its
locks. The protocol, however, does not take into
consideration the semantics of the methods at higher
levels. It also does not consider recovery and the basic
model is less general in the sense that the correctness
condition is appropriate only with an update-in-place
single version implementation of objects.

- In [48], a locking protocol is given where the authors
assume a notion of commutativity in which the conflicts
between lower level methods are ignored due to
the commutativity of the corresponding higher level
methods in the nested execution. This protocol
can exploit the semantics of methods invoked on
encapsulated objects, and unlike previous work on
concurrency control for abstract data types, it takes care
of the fact that such data types can be implemented
in terms of other ADTs. To ensure that each
method execution appears as an indivisible action,
the dynamic method invocation hierarchies that result
from transaction executions are treated as open nested
transactions. In such transactions, the locks of a
subtransaction are released when the subtransaction
completes and only a semantic lock is held further
by the parent of the subtransaction. However, their
protocol does not address the problem of referentially
shared objects (non-disjoint complex objects) in object-
oriented databases. This is a limitation of the algorithm
as composition is a fundamental property of object-
oriented databases as well as a necessary condition
for modular design. Such referential sharing cannot
be described statically since it may arise dynamically
depending on the input parameters and the specific
execution environment. This protocol requires locks
at both for the atomic operations as well as for
method executions. Thus, the system designer has to
derive a conflict or commutativity table not only for
the atomic operations of each class, but also of all
method executions of all the classes in the database.
Furthermore, their approach does not model crash
recovery.

- In [49], another concurrency control protocol that
provides a uniform treatment for dealing with both
class and instance objects has been proposed. Their
approach extends the two-phase locking protocol to
objects with arbitrary operations. This protocol
allows sharing of locks between conflicting operations
provided that the order between the operations is
preserved. The protocol is also extended for
synchronizing nested executions of transactions on
objects. By allowing updates to both the classes
and instances of these classes, this protocol eases the
problem of extensibility. The drawback of the scheme
is its inability to exploit the semantics of higher level
methods to increase concurrency.

- Another semantic-based locking protocol is presented
in [50]. It generalizes Moss’s original nested
transaction protocol by incorporating rich semantics
information available in object-oriented databases.
Their method incorporates the notion of complex
objects, nested executions, and dynamic conflicts
resulting from referentially shared objects. The
correctness of their protocol is based on the theory
of nested transactions proposed in [20]. Their
method uses commutative relations that derived from
the semantics of methods of classes. The conflict
and commutativity relations between methods that
referentially share sub-objects are left unspecified and
are determined dynamically during execution of such
methods. Thus, tests for conflict are done only for
methods executing at the same objects. Methods
on different objects are executed concurrently and,
depending on their executions, the protocol will capture
the conflict relations among such methods. The
protocol depends on the assumption that objects are
designed modularly and are hierarchically organized in
a static manner. However, object-oriented databases
may be designed to violate this restriction. Also,
recovery issues have been ignored.

- The open nested transaction model has also been
implemented in mobile databases in [6] for modelling
mobile transactions as a set of subtransactions. The
model allows for disconnected operation by
supporting unilateral commitment of subtransactions
and compensating subtransactions. Further, the model
can be customized based on the application by
varying the degree of isolation and autonomy of the
subtransactions of mobile transactions.

- The multi-level transaction model given by Weikum
has also been used recently in [51] to efficiently
maintain the replicated data and materialized views by parallelizing the execution of updates.

- In the nested process management environment [7], a workflow process is defined using a hierarchical collection of transactions. The model utilizes the relaxed notion of isolation and atomicity within nested transaction structures. Failure handling is supported by using a two-phase approach. During the first phase of recovery, a bottom-up lookup along the task tree is performed to determine the oldest parent transaction that does not need to be compensated. The next phase involves compensation of all the children of this parent. In this mode, failure atomicity of the workflow is relaxed in terms of compensating only parts of the workflow hierarchy.

5. NESTED TRANSACTION RECOVERY ALGORITHMS

Issues of recovery have been extended to include nesting of transactions [2, 8, 20]. Most of the recovery algorithms available for nested transactions are variations of Moss’s model described in [2]. The model uses only normal read and write operations. Each transaction has to acquire the respective lock before accessing a data object. The model is based on nesting where a subtransaction’s effect cannot be seen outside its parent’s view. Briefly, the algorithm is as follows: when a transaction writes a data object, a new version of the object is created. This version of the object is stored in volatile memory. When the subtransaction commits, the updated versions of the object are passed to its parent. If the transaction aborts, the new version of the object directly or indirectly modified by the transaction is discarded. When the top level transaction commits, the current version of each object directly or indirectly modified by this transaction is saved in stable storage.

- The intentions-list and undo-logging recovery algorithms given in [30, 52, 53] are generalized in [28, 29, 35] to handle recovery in the nested transaction environment by exploiting the commutative properties of the operations. The intentions-list algorithm works by maintaining a list of operations for each transaction. When a transaction commits, its list is appended to its parent; when it aborts, the intentions list is discarded. Also, before executing an operation, say \( p \), of a transaction, the intentions lists for all transactions are first checked to make sure that none of the operations already executed conflicts with \( p \). That is, if \( p \) commutes with operations of all other concurrent transactions then the operation \( p \) is executed; otherwise, it is blocked. When the top level transaction commits, its intentions list is transferred to the log. This scheme provides recovery from transaction aborts only and does not handle system crashes.

- The undo-logging algorithm has also been presented in [35] within the same framework as discussed above. In a nested transaction system, a transaction \( T' \) that is not an ancestor of \( T \) can affect \( T \) by accessing an object which is later accessed by \( T \). If there are intervening commitment actions for all ancestors of \( T' \) up to the level of least common ancestor of \( T \) and \( T' \) then only the committed updates of \( T' \) are visible to \( T \). This helps in avoiding cascading aborts. This strict execution property of transactions leads to the concept of visibility in nested transactions: a transaction \( T' \) is visible to another transaction \( T \) if a commit action for every ancestor of \( T' \) up to (but not necessarily including) the least common ancestor of \( T \) and \( T' \) precedes \( T \). In order to increase concurrency, the undo-logging recovery scheme relaxes the visibility condition by allowing some non-strict executions. It allows a transaction to share the uncommitted updates made by other transactions by exploiting commutativity of operations; an operation of a transaction \( T \) is considered executable only if it commutes with all operations executed by transactions that are not visible to \( T \). On execution of an operation, the data object record changes in its state and the new state is transferred to the log. When a transaction aborts, in contrast to the intentions-list algorithm, all operations executed by its descendants on the object are undone from its current state and are also subsequently removed from the log. This algorithm does not take care of recovery from system crashes. In both intentions-list and undo-logging algorithms, an incomplete transaction is allowed to make uncommitted updates available to those transactions that perform a commutative operation. However, this is restricted to transactions at the same level of abstraction. This limits availability. In both algorithms, all the work done by descendant transactions are discarded in case of aborts at higher levels. This may not be possible or desirable in many real time applications.

- The above two recovery models consider semantics of operations at leaf level only. System R [10] exploits layer-specific semantics but restricted to two levels of transaction nesting. In System R, to perform recovery, updates are undone by performing inverse tuple-level operations. For this purpose, System R records tuple updates on a log. To recover from a system crash, before applying any tuple-level log record, the database must first be restored to some tuple-level consistent state. In other words, a low-level recover mechanism is necessary to make tuple actions appear atomic.

- WeiKum presented a multi-level transaction model [21, 25] at each level of operation nesting, as discussed before. The recovery algorithm for multi-level transactions in [21] and [25] provides inter-transaction recovery in the following way: when a high level transaction aborts, its effect is undone by executing an inverse action that compensates [54] the completed transaction rather than backing the subtransaction out. Recovery from system crashes is provided by executing undo actions at the upper levels and redo actions at the leaf level. Each level is provided with a level-specific recovery mechanism. The algorithm
achieves non-strict executions by exploiting operation semantics of transactions at each level. However, the operation semantics of transactions at each level may not always be easy to exploit, i.e. the various levels of data abstraction may not be known. Also, in this model, transaction aborts are no longer implemented by restoring the pre-transaction state. This is not desirable in many real time database applications though. This model increases concurrency as compared to a single level algorithm, however, the major drawback of the algorithm is the possibility of excessive log I/O at each level of nesting.

- In [55], a recovery method for multi-level systems is presented using multi-level nested transactions. The algorithm uses ARIES/NT [56] and has the same desirable features.

- In [57], a crash recovery technique similar to shadow page has been suggested in a nested transaction environment based on undo/redo log methods. However, the underlying transaction model is the same as before. In terms of logging, both undo and redo logs need to be logged.

- Mohan and co-workers [56, 58] have discussed ‘write-ahead logging’, a crash recovery algorithm based on using a conventional nested transaction model. This ‘undo, redo’ type of recovery model exploits semantics of nested transactions. The actions of a transaction undone during a previous abort have not been undone again in case of one more failure. This is an advantage over Weikum’s multi-level algorithm since it requires undo actions to be undone again in case of one more failure.

- In [59, 60], an open and safe nested transaction model is introduced to deal with long transactions. Prewrite operations [61] have been used in a nested transaction environment to increase concurrency. The notion of a recovery-point subtransaction has been introduced. In a nested transaction tree, if a recovery-point subtransaction executed successfully, its effects are not discarded. The recovery-point subtransaction can release the locks before the commitment of its ancestors, and hence increases concurrency. The model is open as it allows early reads (before actual writes) and is safe as no undo actions are required in case of a system crash. The system’s restart and buffer management operations are also themselves modelled as nested transactions to exploit possible concurrency during restart operations. The crash recovery model has been designed and a correctness proof is carried out using an I/O automaton model [8].

6. FUTURE RESEARCH DIRECTIONS

Some of the open research issues in nested transaction models are as follows:

- Several research papers have been presented on concurrency control issues in nested transaction environments but very little has been reported on recovery algorithms. Recovery algorithms need to be designed especially in the environment of object-oriented databases using nested transactions. In all the versions of the nested transaction recovery models discussed, if a transaction aborts, all its descendants’ effects are discarded either by removing the operations, by restoring pre-images or by executing compensatory undo actions. Also, in most models, the effect of a committed subtransaction is made permanent only when its top level transaction commits. In many real time applications, it is unacceptable that the work of a long-lived transaction is completely undone by using either of these two techniques if the transaction eventually fails. Also, in most of the nested transaction models in order to increase concurrency, the non-strict execution is provided by exploiting the commutative properties of the semantics of operations at the same level of data abstraction. In such models, the leaf level locks are released early only if the semantics of the operations are known and the corresponding compensatory actions defined. However, in some applications, the semantics of transactions may not be known and hence, it is difficult to provide non-strict executions. Therefore, new recovery and concurrency control algorithms are also needed in such environments. The work reported in [59, 60] addresses some of the above issues. Recovery protocols should also minimize the impact of recovery activities during normal transaction processing by handling fast transaction aborts and system restart. More work is needed in this direction.

- Another important area is to do the performance evaluation of nested transaction concurrency control and recovery algorithms to investigate the simulation parameters for the nested transactions which are not known. Since performance analysis can reflect the cost associated with the techniques such as complexity and the overhead incurred by the technique, the research in this direction can be helpful in reducing the concurrency and recovery cost.

- Another direction of future work should focus on the development of semantics-based concurrency control algorithms in nested transaction models by relaxing the serializability requirement and failure atomicity. Also, nested transaction-based locking protocols need to be developed so as to exploit the semantics of higher level methods in OODBMS. These new algorithms are to be extended in distributed and parallel database environments.

- Another research area of interest is to develop the nested transaction recovery algorithms in the environment of knowledge base systems [39]. In [39], only the concurrency control algorithm is given. The recovery algorithm in a knowledge base system is still an open issue.

- There are various complex algorithms and areas of transaction processing systems that would benefit from a more rigorous analysis within the framework of the
nested transaction model [8] and the I/O automaton model [36]. The level of detail in these algorithms also makes careful reasoning about these algorithms very difficult. Many of these algorithms are related to data replications [62, 63], and are concerned with transaction processing using some sophisticated data structures [64]. The detail in these algorithms is not enough to understand the working and to reason about the correctness of the algorithms. In these algorithms, the conditions on which the transactions are initiated and how their initiation is controlled by the scheduler are not taken into account explicitly. These details will help in developing a deeper insight about the working of different modules of the algorithms necessary for the implementation of complex systems. The main reason for these drawbacks is that transactions, objects and the scheduler are not modelled explicitly. Moreover, the correctness proof does not give much insight into the operations of different modules of the algorithm. The algorithms also ignore recovery from transaction aborts.

- In the case of the virtual partition algorithm [62, 63], the algorithm allows some read-only transactions to ‘run as if in the past’, and therefore it does not satisfy the external consistency condition since the apparent serial order has a different sequence when projected on the timestamp order. Hence, it does not satisfy the serial correctness condition as defined in [43]. That is, the virtual partition algorithm cannot be proved to be ‘serially correct’ in the sense of [43]. Therefore, new correctness conditions have to be defined.

- Attention is to be given to designing concurrency control and recovery algorithms that take advantage of the knowledge of particular data structures and the semantics of operations such as insert, delete, find, etc. to improve availability and expedite accesses. Data structures which have been analysed with the above in mind are the B-tree and its variants [45, 65], extendible hashing [66, 67], linear hashing [64, 68, 69], exponential hashing [70], and dynamic hashing [71, 72]. New algorithms need to be developed using various hash structures in the environment of nested transactions in order to further increase concurrency and to provide efficient recovery. The new algorithms should also remove the drawbacks of the work on B-trees [44] as discussed earlier. Another direction of research is in the area of dealing with crash recovery in the nested transaction environment using hash structures and B-trees.

7. CONCLUSION

In this paper, we have reviewed existing work in the area of concurrency control and recovery algorithms in nested transaction environments. We have presented the research ideas in a concise yet informative way so that readers will be well informed on the nature of work done on nested transaction management. We have discussed the techniques that have been developed and analysed them with respect to their main features, and by highlighting their merits and demerits. It is evident from our discussions that performance evaluation of concurrency control and recovery algorithms in nested transaction environments is still an open area which can be considered for future research. There is a need to develop recovery algorithms involving various data structures and nested transactions. We need some simulation studies to determine how well the recovery and concurrency control algorithms in nested transaction environments work. We have discussed some more open problems in the area of nested transaction concurrency control and recovery algorithms in Section 6.

ACKNOWLEDGEMENTS

We would like to thank anonymous referees for their suggestions and comments to improve the contents and presentation.

REFERENCES

et al.


Encina at http://www.transarc.com/afs/ProdServ/Product/Encina.


