A Concurrency Control Algorithm for an Open and Safe Nested Transaction Model

Sanjay Kumar Madria
Missouri University of Science and Technology, madrias@mst.edu

International Conference on
Information, Communications and Signal Processing
ICICS '97
Singapore, 9-12 September 1997

A Concurrency Control Algorithm for an Open and Safe Nested Transaction Model

Sanjay Kumar Madria

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

Abstract

In this paper, we present a concurrency control algorithm for an open and safe nested transaction model. We use prewrite operations [19] in our model to increase the concurrency. Prewrite operations are modeled as subtransactions in the nested transaction tree. The subtransaction which initiates prewrite subtransactions are modelled as recovery point subtransaction [23]. The recovery point subtransaction can release their locks before its ancestors commit. Thus, our model increases concurrency in comparison to other nested transaction models. Our model will be useful in the environment of long-running transactions common in object oriented databases, computer aided design and software development process.

1 Introduction

In a nested transaction model [18], a subtransaction 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. This parent-child relationship defines a nested transaction tree, and such transactions are termed as nested transactions. Failure of subtransactions may result in invocation of alternate subtransactions that could replace the failed ones to accomplish the successful completion of the whole transaction. A child transaction has access to the data locked by its parent. It is atomic with respect to its parent and its siblings. It is serializable with its siblings. It becomes permanent only if its parent becomes permanent. If a parent aborts, all its descendants' effects are to be undone. Therefore, a child's scope is restricted to its parent only. Hence, this model is termed as closed nested transaction model. A parent commits only after all its children are terminated.

In [17], Lynch has presented a complete proof of the exclusive locking algorithm for nested transactions. Reed [26] has presented a multi-version timestamp concurrency control algorithm to provide nested transaction based data management. In [1], a formal analysis of the algorithm is given. Moss [18] has extended two phase locking with separate read/write locks to handle nesting. A formal version of this algorithm appeared in [5]. In [7], the read-update locking algorithm [29] has been generalized and a new commutative locking algorithm has been introduced to handle nested transactions. Fekete et al. [4] have presented a serialization graph construction for nested transactions. The quorum consensus algorithm for data replication is generalized by Goldman in [8] to accommodate nested transactions. The multi-granularity algorithm has been extended to nested transaction systems in [13]. Nested transactions have also been discussed in the context of B-Trees [3] and linear hash structures [22]. Some more related work appears in [12,14]. Most of the above mentioned algorithms are discussed using I/O automaton model [16]. Many of these algorithms appear in [4].

Nesting in transactions corresponds either to the nesting of procedures or to the nesting of layers of data abstractions. In the first kind [15,18]), a subtransaction's updates are not visible outside its parent and therefore, availability is restricted. If the parent aborts, the subtransaction is also aborted. In the second kind [2,28,30], 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. Since basic (i.e., read and writes) locks are released early and have possibly been acquired by other *transactions, an abort has to take place in the* form of compensatory operation.

A related but more complex notion of nesting emphasizing level of data abstraction has been studied in [2,21,30]. To exploit layer specific semantics at each level of operation nesting, Weikum presented a multi-level transaction model [30] called open nested transaction model. 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. In this model, a subtransaction is allowed to release locks on finishing before the commit of higher level transactions. In case a higher level transaction aborts, the aborted transaction's effect is to be undone by compensatory transaction. This model has also been studied in the framework of object oriented databases in [25, 27].

In the closed nested transaction model, the availability is restricted as the scope of each subtransaction is restricted to its parent only. This forces a subtransaction to pass all its locks and versions of data objects updated to its parent on commit. The effect of a committed subtransaction is made permanent only when its top level transaction commits. In many applications, it is unacceptable that the work of a long-lived transaction (common in engineering design applications [9,10] is completely undone by in case transaction eventually fails at finishing stage. The current strategy forces short-lived transactions to wait to acquire their locks until top level transactions commit and release their locks. Therefore, the model is not appropriate for the system that consists of long and short transactions.

In the open nested transaction model, the leaf level locks are released early only if the semantics of the operations are known and the corresponding compensatory actions defined at each level. However, the semantics of transactions may not be known and not all actions may be compensatable (e.g., handing over a cheque). In real time situations, there are other classes of operations that have an irreversible external effect, such as handing over huge amounts of money at an automatic teller machine (ATM). Such operations have to be deferred until the top level transaction commits, which restricts availability.

There are two basic motivations behind our open and safe new nested transaction model presented in [19,23]. First, it is desirable that long-lived transactions should be able to release their locks before top-level transactions commit. Second, it may not be desirable or possible to undo or compensate the effects of one or more of the important committed descendants after the failure of a higher level transaction due to an abort or a system crash. We have presented a crash recovery algorithm of our model in [23]. Our model allows some particular sub-transactions to release their locks before their ancestor transactions commit. This allows the other subtransactions to acquire required locks earlier. Our model handles the situations where a committed lower level subtransaction's effect cannot be undone or compensated in case of a higher level transaction's failure. It is possible that a transaction's semantics may be such that beyond a certain point, either it cannot rollback entirely or its effect should not be compensated. We introduced the concept of a "recovery point subtransaction" [23] of a top-level transaction in a nested transaction tree. It is essentially a subtransaction after the commit of which its ancestors are not allowed to rollback. In other words, once the recovery point subtransaction of a top level transaction has committed, all its superior transactions are forced to commit. In case, it aborts, its ancestors can choose an alternate path to complete their execution. It says that recovery point subtransaction's commit (e.g. mailing a cheque) is crucial for the commit of its ancestors. In case a superior transaction aborts or system fails after the commit of its recovery point subtransaction, the failed transaction has to complete on system revival. Such a transaction execution permits a recovery point subtransaction to reveal its result to other transactions at any level of nesting before its superior transactions commit. A recovery point subtransaction's effect is made durable before its top level transaction's commit. This results in relaxation of the isolation property of the transaction.

In our model, to avoid undo actions and the consequent cascading aborts as well as to increase the availability , we assumed that each write issues a prewrite operation [19,23] for the objects it intends to write. Each prewrite operation contains the value that a user transaction wants to write and precedes the associated final write. A prewrite operation actually does not change a data object's state but only announces the value the data object will have after the associated write is performed. In response to a read operation, each DM returns the prewrite value (if any) otherwise it returns the write value. The advantage of prewrite is that a read operation can get the value before a data object's state is changed. Hence, this results in increasing the availability further with reduced execution time. Prewrite operations are particularly helpful in the engineering design applications [9,10] and in large software design projects [11] etc. where transactions are long.

In our nested transaction model [19], a subtransaction that initiates different prewrite access subtransactions at leaf level for different data objects is defined to be the recovery point subtransaction. These announced prewrite values are made visible to other subtransactions after the commit of the recovery point subtransaction. The prewrite subtransactions release their locks before their ancestors commit. Discarding some of the prewrites before the commit of the recovery point subtransaction will not introduce cascading aborts since the prewrite values are made visible only after the commit of the recovery point subtransaction.

In this paper, we will formally design our nested transaction model and discuss the concurrency control algorithm for our model.

2 Nested Transaction Model and System Configuration

Our nested transaction system model consists of transaction managers (TMs), recovery managers (RMs) and data managers (DMs). The data objects are modeled by the data managers (DMs). Each data manager keeps a copy of the data object in the secondary storage, called stable-db. The prewrite and write values of each object are kept in the respective buffers at the corresponding DMs. These are called prewrite- and write-buffers, respectively. Physically, only a subset of these DMs will have prewrite and write values of the data objects in the corresponding buffers. A read operation gets the value of the referenced data object from the prewrite-buffer (if any) otherwise it gets the value from the write-buffer. If the DM does not even have a copy of the data object in the write-buffer, the read operation gets the value from the stable-db copy of the data object. The write-buffer's contents of a data object are transferred periodically to stable storage.

Our model has two transaction managers (TMs) for performing read (read-TM), write (write-TM) and of these, read- and write-TMs are initiated by the user transactions. A hidden daemon transaction is associated with each write-TM to co-ordinate the buffer management operations; i.e., the transfer of a data object's value to stable-db during the normal operations. To achieve the notion of spontaneity and transparency of buffer management operation, the daemon transaction wakes up and commits with respect to its associated transaction. Therefore, during the span of daemon transaction, a daemon can initiate many transfer-RMs.

TMs are situated at one level below user transactions. Next level of transaction hierarchy has six different recovery managers for co-ordinating read (read-RM), prewrite (prewrite-RM), write (write-RM), transfer (transfer-RM) These RMs initiate access subtransactions situated at the leaf level. Each read-, prewrite- and write-RM initiates read, prewrite and write access subtransactions, respectively. A read access reads the value either from the prewrite- or the write-buffer of the data object whereas a write subtransaction accesses only the write-buffer component of the data object. The nested transaction tree structure is shown in figure.1.

We assume that each user transaction knows its write-set before initiating a write-TM in order to write all the data objects in its write-set. A write-TM first initiates a prewrite-RM which further initiates prewrite access subtransactions in order to announce prewrites for all the data objects contained in the write-set. This value for each data object is written in the prewrite-buffer allocated in the volatile memory. Modeling prewrites at leaf level provides user transparency to the prewrite operations.

We formally specify the prewrite-RM as the recovery point subtransaction of the top level transaction. Once the prewrite-RM has committed, the prewrite values become visible outside its parent's view at any level of nesting without necessarily requiring the commit of all its superior transactions. After the prewrite-RM's commit, the write-TM initiates a write-RM to update all the data objects whose prewrite values have been announced before. The final updates are written in the write-buffers allocated in the volatile memory at each DM. With the invocation of each write-TM automaton, a daemon transaction is made active automatically which further initiates transfer-RMs. A transfer-RM initiates a transfer access subtransaction to transfer the write-buffer's value to the stable-db. The write-buffer's contents can be transferred without the commit of the top level transaction because write-values, once written, cannot be undone or lost.

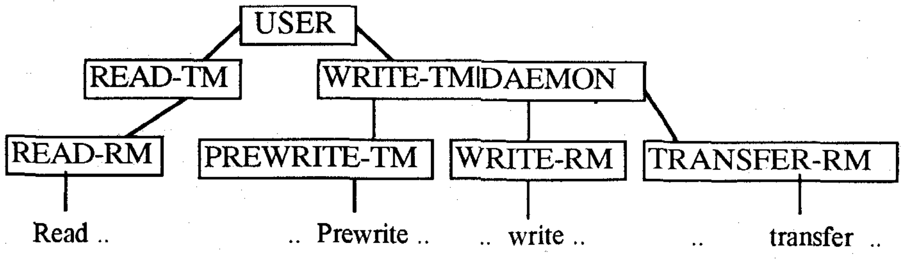

Figure 1

3 Concurrency Control and Locking

In this section, we mainly discuss the type of conflicts which occur in our model during normal operations and the locks needed to control them.

A read access transaction can read the value from the prewrite-buffer if it has the prewritten value of the corresponding data object. Otherwise, it gets the value from the write-buffer. A transfer access also reads a write-buffer's value to transfer it to the stable-db. The prewrite and write access subtransactions access prewrite- and write-buffers, respectively.

A prewrite operation introduces some more conflicting operations apart from usual read-write and write-write conflicts. The three pairs of conflicting operations are prewrite-prewrite, prewrite-write and read-prewrite (only if read returns the prewrite-buffer's value). A prewrite-prewrite conflict occurs due to the fact that a prewrite value cannot be changed unless its associated write is performed. When a write operation is operating, its associated prewrite value cannot be changed or vice-versa. This is because a write operation replaces the value of the data object in the write-buffer by the prewrite-buffer's value. Hence, the prewrite and write operations cannot be executed concurrently. Also, while reading a data object's prewrite value, no other prewrite access transaction can change its prewrite value or vice-versa. Otherwise, a read operation can get an inconsistent value. A transfer access is similar to a read access and hence, it does not introduce any new pair of conflicting operations.

To control concurrent read, prewrite, write and transfer accesses, each access subtransaction has to acquire its respective lock before accessing a data object. Our algorithm uses a read-lock for read and transfer access operations, and write- and prewrite-locks for write and prewrite access operations, respectively. Prewrite- and write-locks are exclusive locks whereas a read-lock is a shared lock.

In our locking protocol, a transaction may hold and retain locks. A transaction holding a lock for an object is allowed to access the corresponding object. The object is not allowed to access if a transaction only retains the lock. Since the accesses are situated at leaf level only, all the holders of locks are at leaf level only. A retained lock is only a place holder indicating that transactions outside the hierarchy of the retainer cannot acquire the same lock or any of its conflicting locks but descendants of the retainer can acquire the same or non-conflicting locks.

Whenever a prewrite access transaction commits, it passes its prewrite-lock to its parent transaction (prewrite-RM). The prewrite-lock is passed to the parent so that the transactions outside the parent's hierarchy cannot get the prewrite-lock. This is because in case an upper level transaction of the committed prewrite access subtransaction aborts or system fails, the prewrite value is to be discarded. However, whenever the recovery point subtransaction (prewrite-RM) commits, the committed subtransaction's lock is passed directly to the least common ancestor (of other waiting accesses and the committed transaction) without necessarily the commit of all its superior transactions upto the least common ancestor (l.c.a.). This also holds in case of commit of recovery point subtransaction's superior transactions. The l.c.a. of waiting accesses and of committed transaction is determined dynamically at run time. The prewrite-lock cannot be released entirely because a new prewrite operation for the same data object cannot be initiated unless the write operation corresponding to the last prewrite is committed.

Locks inherited by a l.c.a. enables waiting read and write access subtransactions to acquire their respective locks early. This increases the availability further since a waiting access subtransaction T_2 can get its respective lock before all the ancestors of the committed transaction T_1 upto the least common ancestor of T_1 and T_2 are necessarily committed. For similar reasons as stated before, whenever a write access transaction commits, its lock is also passed to the least common ancestor of other waiting access subtransactions to enable them to acquire their respective locks. This also holds for all the ancestors of the committed write access subtransactions in case of their commit. Passing the lock early to l.c.a. is safe because once the recovery point subtransaction is committed, its higher level transactions have to commit. It seems that once a transaction is committed, its write-lock can be released entirely but this is not desirable for the following reasons. First, a transaction cannot get a write-lock unless its ancestors hold prewrite- and read-locks for the object. Therefore, releasing a write-lock does not help unless the other conflicting locks are released. Second, in case of restart, some transactions might need the write-locks to complete their interrupted execution due to failure.

The prewrite-lock held by a prewrite access subtransaction is released in case it, or any of its ancestors, aborts or system fails because its effect is to be discarded. However, when a write access subtransaction aborts after it acquires the write-lock, its lock is passed to its parent. Furthermore, when a transaction aborts, the write-lock held by any of its descendants is inherited by the parent of the aborted transaction. In effect, the write- and/or prewrite-lock held by any transaction (except by prewrite access subtransactions) is not released entirely in case the lock retaining transaction aborts or system crashes. Releasing a prewrite- or a write-lock entirely in case of aborts or system crash may create an inconsistent state. This is due to the fact that some subtransactions might have to complete their remaining execution on revival. Read-only transactions can release their lock entirely in case of their aborts and pass their locks to parent transactions on commit. A transfer access releases its lock to the daemon transaction on commit as its effects are not going to be discarded in case of aborts at higher level or system crash. In case of a transaction abort in the hierarchy of daemon transactions, its lock is released entirely. The daemon transaction releases its lock according to its associated transaction.

Formally, we have the following locking rules :

1. A read access can acquire a read-lock only if the prewrite-lock and write-lock on the corresponding DM is retained by its ancestor transaction.
2. A prewrite access subtransaction can get its respective prewrite-lock only if the prewrite-, write- and read-locks are retained by its ancestor transactions.
3. A write access transaction can get its write-lock only if read-, prewrite-, write-locks are held by its ancestors.
4. A transfer access can get a read-lock only if the write-lock is held by its ancestors. Note that this rule is similar to rule 1.
5. When a read access subtransaction commits, it releases its lock to its parent. When its parent commits, it passes the lock to its parent and so on.
6. When a transfer access transaction commits, it passes its lock to the daemon transaction. When a daemon commits, it passes its lock according to its associated transaction.
7. When a prewrite access commits, it passes its lock to its parent. When its parent commits (prewrite-RM), it passes the lock to the least common ancestor of all other access descendants waiting for the lock.
8. When a write access transaction commits, it releases its lock to the least common ancestor of all other access descendants waiting for the lock.
9. When a write access or prewrite-RM or any of its ancestor transaction aborts, the aborted transaction's locks are passed to its parent transaction.
10. When a read-only transaction aborts, its lock is released entirely. When a transaction in the hierarchy of daemon transaction aborts, its lock is released entirely.

4 Conclusion

In this paper, we have presented a concurrency control algorithm for an opne and safe nested transaction model. Our algorithm increases concurrency as compared to other models as it allows early release of locks by some subtransactions. We have introduced prewrite operations in our model to increases the concurrency and to avoid undo actions on transaction aborts. We are in the process of proving the correctness of this algorithm using I/O automaton model.

References

[1] Aspnes, J., Fekete, A., Lynch, N., Merrit, M., and Weihl, W., A Theory of Timestamp Based Concurrency Control for Nested Transactions, In Proceedings of 14th International Conference on Very Large Databases, pp. 431-444, Aug., 1988.

[2] Beeri, C., Bernstein, P.A. and Goodman, N., A Model for Concurrency in Nested Transaction System, Journal of the ACM, Vol. 36, No. 1, 1989.

[3] Fu, A. and Kameda, T., Concurrency Control of Nested Transactions Accessing B-trees, In proceedings of 8th ACM Symposium on Priniciples od Database Systems, pp. 270-285, 1989.

[4] Fekete, A., Lynch, N., Merrit, M. and Whiel, W., Atomic Transactions, Morgan-Kaufmann, 1993.

[5] Fekete, A., Lynch, N., Merrit, M. and Whiel, W., Nested Transactions and Read/Write Locking, In Proceedings of the 6th ACM Symposium on Principles of Database Systems, pp. 97-111, San diego, CA, 1987.

[6] Fekete, A., Lynch, N., Merrit, M. and Weihl, W.E., Commutativity-Based Locking for Nested Transactions, Journal of System Sciences, Vol. 41, No. 1, pp. 65-156, Aug., 1990.

[7] Fekete, A., Lynch, N., and Weihl, W.E., A Serialization Graph Construction for Nested Transactions, In proceedings of ACM Symposium on Principles of Database Systems, 1990.

[8] Goldman, K. and Lynch, N., Nested Transactions and Quorum Consensus, ACM TODS, Dec.1994.

[9] Korth, H.F., Kim, W., Bancilhon, On Long-Duration CAD Transactions, Information Science, 46, pp.73-107, Oct.1990.

[10] Kim, W., Lorie, R., Mcnabb, D. and plouffe, W., A Transaction Mechanism for Engineering Design Databases, in Proceedings of the 10th International Conference on Very Large Databases, VLDB Endowment, pp. 355-362, 1984.

[11] Korth, H.F., and Speegle, G., Long Duration Transactions in Software Design Projects, in 6th International Conference on Data Engineering, IEEE, New York, pp.568 - 574, 1990.

[12] Lee, J.K., Precision Locking for Nested Transaction Systems, Second International Conference on Information and Knowledge Management (CIKM'93), Nov.1993.

[13] Lee, J.K. and Fekete, A., Multi-granularity Locking for Nested Transaction Systems, In proceedings of MFDBS'91, pp. 160-172, Lecture notes in Computer Science, 495, Springer Verlag, 1991.

[14] Lee, J.K. and Fekete, A., Predicate Locking for Nested Transaction Systems, In proceedings of Australian Database Research Conference, pp.217-231, Feb.1992.

[15] Liskov, B., Distributed Computing in Argus, Communication of ACM, Vol. 31, No. 3, pp. 300-312, March, 1988.

[16] Lynch, N. and Merrit, M., Introduction to the Theory of Nested Transactions, Theoretical Computer Science, Vol. 62, pp. 123-185, 1988.

[17] Lynch, N., Concurrency Control for Resilient Nested Transactions, Advances in Computing Research, Vol. 3, pp. 335-376, 1986.

[18] Moss, J.E.B., Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. Thesis. Also, Technical Report MIT/LCS/TR-260 MIT Laboratory for Computer Science, Cambridge, MA., April, 1981.

[19] Madria, S.K., Concurrency Control and Recovery Algorithms in Nested Transaction Environment and Their Proofs of Correctness, Ph.D. Thesis, Department of Mathematics, Indian Institute of Technology, Delhi, 1995.

[20] Madria, S.K. and Bhargava, B., System Defined Prewrites to Increase Concurrency in Databases, accepted for First East-Europian Symposium on Advances in Databases and Information Systems (in co-operation with ACM-SIGMOD), St.-Petersburg (Russia), Sept.97.

[21] Moss, J., Griffith, N. and Graham, M., Abstraction in Concurrency Control and Recovery Management (revised), Technical Report COINS 86.20, University of Massachusetts at Amberest, May, 1986.

[22] Madria, S.K., Maheshwari, S.N. and B. Chandra, Formalization of Liear Hash Structures using Nested Transactions and I/O Automaton Model, communicated to ASIAN'97.

[23] Madria, S.K., Maheshwari, S.N, Chandra, B., Bhargava, B., Crash Recovery Algorithm in an Open and Safe Nested Transaction Model, 8th International Conference on Database and Expert System Applications (DEXA'97), France, Sept.97, Lecture Notes in Computer Science, Springer Verlag, 1997.

[24] Muth, P., Rakow, T.C., Weikum, G., Brossler, P., Hasse, C., Semantic Concurrency Control in Object-Oriented Database Systems, In proceedings of the 9th International Conference on Data Engineering, pp. 233-242, 1993.

[25] Reed, D.P., Naming and Synchronization in a Decentralized Computer System, Ph.D. Thesis. Also, Technical Report MIT/LCS/TR-205, MIT Laboratory for Computer Science, MA., 1978.

[26] Resende, Rodolfo F., Agrawal, D., Abbadi, Amr El, Semantic Locking in Object Oriented Database Systems, Technical Report TRCS 94-01, University of California at Santa Barbara, 1994.

[27] Weihl, W.E., Specifications and Implimentation of Atomic Data Types, Ph.D. Thesis. Also, Technical Report MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA., March, 1984.

[28] Weihl, W.E., Commutativity-Based Concurrency Control for Abstract Data Types, IEEE Transaction on Computers, Vol. 37, No. 12, Dec., 1988.

[29] Weikum, G., Principles and Realization Strategies of Multi-Level Transaction Management, ACM Transaction on Database System, Vol. 16, No. 1, March, 1991.