Partial Strictness in Two-Phase Locking

Eljas Soisalon-Soininen and Tatu Ylönen

Department of Computer Science, Helsinki University of Technology
Otakaari 1, FIN–02150 Espoo, Finland
e-mail: ess@cs.hut.fi, ylo@cs.hut.fi
telefax: +358-0-451 3293, tel: +358-0-4511

Abstract. Two-phase locking is a standard method for managing concurrent transactions in database systems. In order to guarantee good recovery properties, two-phase locking should be strict, meaning that locks can be released only after the transaction's commit or abort. In this paper we show that even exclusive locks can be released immediately after the *commit request* has arrived, without sacrificing any important recovery properties. This optimization is especially useful if the commit operation takes much time compared with the other actions, as for main-memory databases, or if the commits are performed in batches.

1 Introduction

When several transactions operate concurrently on the same database, a mechanism is needed for controlling the concurrency in order to guarantee correctness. That is, we have to ensure that when transactions are run concurrently the consistency of the database is preserved as if the transactions would be run serially. As to concurrency, the formal correctness is based on the notion of serializability, meaning that the interleaved mode of operation has the same result as when the transactions are run one by one.

The most widely used strategy for scheduling concurrent transactions is based on locking. The scheduler locks a data item before a transaction may access it, and no other transaction may access the data item before the lock has been released. Locks are often divided into exclusive and shared locks; several transactions may hold a shared lock on a data item but when one holds an exclusive lock no other transaction may hold any (shared or exclusive) lock on the same data item. In two-phase locking [4], once the scheduler has released the lock for a transaction, it may not subsequently obtain any more locks for that transaction on any data item. It has been shown [13] that two-phase locking is a most general locking strategy that guarantees serializability when the set of transactions may change dynamically.

In addition to guaranteeing serializability the scheduler must reject interleavings that are not recoverable. That is, if a transaction aborts, we must be able to reconstruct the situation we would have arrived at if the aborted transaction had not existed. Moreover, for practical reasons, it is advisable that aborting one transaction does not cause other transactions to abort, i.e., cascading aborts are

avoided, and that recovering can be based on so called before images of data items [1]. This leads to the requirement that two-phase locking should be strict [1], meaning that no lock may be released before the transaction's commit or abort has been processed.

There are some obvious cases in which transaction processing according to the rules of strict two-phase locking is too restrictive. In main memory databases [2, 3] no disk accesses are needed for normal reads and writes but, because of possible failures, in commit the changes must be stored into a disk file. Thus performing the commit action is considerably more costly than the other actions, and much more concurrency would be allowed when all locks could be released immediately after the *commit request* has arrived. Another situation in which this early releasing of locks would be very beneficial is when the group commit optimization [6, 8] is applied, i.e., several commits are collected and performed as a batch.

In this paper we shall investigate in detail how good recovery properties can be retained in conjunction with early releasing of locks. In particular, we shall show that for transaction failures, caused for example by deadlocks, strict executions can be guaranteed. For aborts caused by system failures no such guarantee can be obtained, but even then a recoverable execution can be achieved.

The results have been derived in a formal model in which we have added a new action, called the commit request action, into the set of possible transaction steps. We require that the commit request action appears as the second last step in transactions that end with the commit action. Our results say that recoverability is always preserved for early releasing of locks. Moreover, for all aborts that are not preceded by the commit request the strict behaviour can be guaranteed. If an abort occurs after the commit request strictness cannot be guaranteed, and we suggest that in such cases all active transactions are aborted. It should be noted that it is a very rare, though possible, to have an abort after the commit request (e.g., due to a system crash). An abort caused by a deadlock or requested by the application cannot occur after the commit request has been received.

The idea of early releasing of locks is not entirely new. Similar ideas have been used in main-memory databases and some prototypes [3, 5, 8, 9]. Other related work includes [12], where algorithms are given for determining when it is safe to unlock entities while the transaction is still active, and [11], where commit ordering is used in heteregeneous distributed databases. However, no formal presentation of the ideas has previously been available.

This paper is organized as follows. Sections 2 and 3 define the transaction and execution model. Section 4 is the main contribution of this paper; partially strict histories are defined, and it is proven that they maintain recoverability and do not cause cascading aborts except in rare situations that can be handled specially. Section 5 defines partially strict two-phase locking, and proves that it only accepts partially strict histories. Implementation alternatives for commit ordering are analyzed. Section 6 concludes the paper.

2 Transaction Model

A *transaction* T is a finite sequence of actions of the form *Read*$[x]$ (denoted $R[x]$) or *Write*$[x]$ (denoted $W[x]$), where x is an element of a set of *data items*. For a given data item x there is at most one $R[x]$ and at most one $W[x]$ in a transaction. A *committed transaction* is a transaction followed by a special symbol c, and an *aborted transaction* is a transaction followed by a special symbol a. In the sequel, by the term transaction we mean a committed or an aborted transaction. By $R_i[x]$, $W_i[x]$, c_i, and a_i we denote actions of transaction T_i.

Notice that our model slightly differs from that of [1]. For simplicity, we define transactions and histories as total orders of steps, but our results can be derived also in the more general model of [1] where they are partial orders.

Let $\tau = \{T_1, \ldots, T_n\}$ be a set of transactions. A *complete history* of τ is an ordering of the actions of all of the transactions of τ, with the actions within each transaction in the prescribed order. In other words [10], a complete history of τ is an element of the *shuffle* of τ meaning the set of all sequences obtained by interleaving the sequences $T_1, T_2, \ldots, T_n$. A *history* is a prefix of a complete history. A complete history is *serial* if, for every two transactions T_i and T_j appearing in H, all actions of T_i precede all actions of T_j, or vice versa.

Let H be a history of $\tau = \{T_1, \ldots, T_n\}$. We say that transaction T_i is *committed* (resp. *aborted*) in H, if c_i (resp. a_i) appears in H. Transaction T_i is *active* in H if it is neither committed nor aborted. Given a history H, the *committed projection of* H, denoted $C(H)$, is the history obtained from H by deleting all actions that do not belong to transactions committed in H. Two actions of H are said to *conflict* if they both are associated with the same data item x and at least one of them is $W[x]$.

The *serialization graph* for history H of τ, denoted $SG(H)$, is a directed graph whose nodes are the transactions of τ that are committed in H and whose edges are $T_i \rightarrow T_j$, $T_i \neq T_j$, where T_i has an action that precedes and conflicts with an action of T_j in H. A history H is called *(conflict-)serializable* if $SG(C(H)) = SG(H_s)$ for some serial history H_s.

3 Recoverable Histories

Let H be a history of a transaction set $\{T_1, T_2, \ldots, T_n\}$. We say that action $R_i[x]$ in T_i *reads from* $W_j[x]$ in T_j in H [1], if H is of the form

$$\ldots W_j[x] \alpha R_i[x] \ldots,$$

where α does not contain a_j or any $W_k[x]$ except possibly when α also contains a_k (the abort action of transaction T_k). We say that transaction T_i *reads from* transaction T_j in H, if for some $R_i[x]$ and $W_j[x]$, $R_i[x]$ reads from $W_j[x]$ in H.

A history H is *not recoverable* [1] if it is of the form

$$\ldots W_j[x] \ldots R_i[x] \ldots c_i \ldots,$$

where $R_i[x]$ reads from $W_j[x]$ and c_j does not occur in H. Otherwise, H is *recoverable*.

In other words, history H is not recoverable if it may be followed by a_j (the abort action of transaction T_j) and the result of transaction T_i is dependent on transaction T_j but T_i has already committed.

Cascading aborts may occur when a transaction reads from another transaction which will be aborted later. We say that a history H *may create cascading aborts* if it is of the form

$$\ldots W_j[x] \ldots R_i[x] \ldots a_j \ldots,$$

where $R_i[x]$ reads from $W_j[x]$, and a_i does not precede a_j in H. Otherwise, H *avoids cascading aborts*.

In other words, history H may create cascading aborts if transaction T_i reads from transaction T_j in H and T_j aborts in H but T_i does not end in H.

A history H is *not strict* if it may create cascading aborts or H is of the form

$$\ldots W_j[x] \ldots W_i[x] \ldots,$$

where transaction T_j has not ended before $W_i[x]$, and some history having H as a prefix contains a_j. That is, it is required that transaction T_i cannot write the same data item as transaction T_j if T_j has not ended yet and may be aborted later. Otherwise, history H is *strict*.

Notice that in practice, of course, an abort may occur for any transaction, and thus the above requirement that transaction T_j may be aborted later seems unnecessary. In the next section, however, we want to distinguish between the case in which an abort is caused by a fairly uncommon system error (such as a system crash) and the case in which the abort is caused by a normal transaction failure. Typical transaction failures come from deadlocks, or when the application wants to abort the transaction.

4 Partially Strict Histories

In this section we want to relax the strictness requirement for histories. The idea is that we want to allow reading and writing from a transaction that has not committed yet but all other actions have been done and, additionally, a request for committing the transaction has arrived. Then, normally, the transaction will commit, and we will prove that the partial strictness obtained in this way will coincide with the true strictness. In practice, in cases of system crash and some unusual errors such as the lack of disk space when writing, we cannot guarantee strict executions. However, these situations are rare, and they can be handled by simply aborting all active transactions. The necessary recoverability will remain in all cases, i.e., even when a commit request is followed by an abort.

In our model, we add a new action c^R to the set of possible actions of transactions. The action c^R meaning the commit request may only occur as the second last action of a transaction. Always when a transaction ends with the action c (commit) it must be preceded by c^R.

We say that a history H is *partially strict* if the following three conditions hold:

(i) whenever H is of the form

$$\ldots W_j[x] \ldots R_i[x] \ldots,$$

where $R_i[x]$ reads from $W_j[x]$, then ${c_j}^R$ precedes $R_i[x]$ in H,

(ii) whenever H is of the form

$$\ldots W_j[x] \ldots W_i[x] \ldots,$$

then a_j or ${c_j}^R$ precedes $W_i[x]$, and

(iii) whenever ${c_i}^R$ precedes ${c_j}^R$ in H, then c_i cannot occur in H without preceding c_j.

Theorem 1. Partially strict histories are recoverable.

Proof. Let H be a partially strict history. For the sake of contradiction assume that H is not recoverable. Then, by definition, H is of the form

$$\ldots W_j[x] \ldots R_i[x] \ldots c_i \ldots,$$

where $R_i[x]$ reads from $W_j[x]$ and c_j does not occur in H. As H is partially strict, we conclude from condition (i) of the definition that ${c_j}^R$ must precede $R_i[x]$. Thus ${c_j}^R$ must precede ${c_i}^R$ implying by condition (iii) that c_j should precede c_i, which is a contradiction. □

Lemma 1. A partially strict history H avoids cascading aborts, provided that a_i cannot follow ${c_i}^R$ in H for any transaction T_i.

Proof. Assume that H may create cascading aborts. Then H is of the form

$$\ldots W_j[x] \ldots R_i[x] \ldots a_j \ldots,$$

where $R_i[x]$ reads from $W_j[x]$, and a_i does not precede a_j in H. Now because $R_i[x]$ reads from $W_j[x]$, we conclude that ${c_j}^R$ must precede $R_i[x]$. Thus, by the assumption of the lemma a_j cannot follow, and we have a contradiction. □

Theorem 2. A partially strict history H is strict, provided that in no continuation of H (which may be H itself) a_i follows ${c_i}^R$ for some transaction T_i.

Proof. Lemma 1 implies that, under the assumption of the theorem, partially strict histories avoid cascading aborts. Then let H be a partially strict history that avoids cascading aborts. If H is not strict, then H is of the form

$$\ldots W_j[x] \ldots W_i[x] \ldots,$$

and there is a history H'

$$\ldots W_j[x] \ldots W_i[x] \ldots a_j$$

that has H as its prefix. Moreover, because H is partially strict, ${c_j}^R$ precedes $W_i[x]$ in H. But by the assumption of the theorem, a_j cannot follow ${c_j}^R$, and we thus conclude that H is strict. □

We have thus shown that partial strictness coincides with strictness if in no execution of transactions the abort action follows the commit request action. This result can be interpreted such that when only transaction failures are present, then the true strictness is obtained through partial strictness. Notice that in the case of a transaction failure such as a deadlock or when the application wants to abort the transaction the commit request action cannot be followed by an abort. This means that under partial strictness property we may well resort to standard recovery strategies when transaction failures occur. For example, they do not cause other transactions to abort, as by Lemma 1 transaction failures do not cascade. System failures must be handled in a different way, and a straightforward solution is to abort all active transactions when a system failure occurs.

5 Partially Strict Two-Phase Locking

In this section we shall show how partial strictness of histories, together with their conflict-serializability [1, 10] can be obtained by two-phase locking.

Let T be a transaction. A *locked transaction* $\bar{T}$ of T is a finite sequence of steps containing all actions of T, and instructions $Rlock[x]$, $Runlock[x]$, $Wlock[x]$, and $Wunlock[x]$ (denoted $Rl[x]$, $Ru[x]$, $Wl[x]$, and $Wu[x]$). Moreover, we assume that $R[x]$ is preceded by $Rl[x]$ or $Wl[x]$ and followed by $Ru[x]$ or $Wu[x]$, respectively, and $W[x]$ is preceded by $Wl[x]$ and followed by $Wu[x]$.

A *locking* of a transaction set $\tau = \{T_1, \ldots, T_n\}$ is a set $\lambda = \{\bar{T}_1, \ldots, \bar{T}_n\}$ of locked transactions, such that the subsequence of $\bar{T}_i$ obtained by deleting all $Rl[x]$, $Ru[x]$, $Wl[x]$, and $Wu[x]$ steps coincides with T_i. If H is a history of τ, then $\bar{H}$ denotes a history of λ that has H as a subsequence. If $\bar{H}$ is a history of λ, the locked version of τ, then H denotes the history of τ obtained from $\bar{H}$ by deleting all lock and unlock instructions.

A history $\bar{H}$ of λ is *legal*, if each $Rl[x]$ is followed by $Ru[x]$ before $Wl[x]$ and each $Wl[x]$ is followed by $Wu[x]$ before another $Wl[x]$ or $Rl[x]$. A history H of τ is *realized* or *accepted* by λ, if there is a legal history $\bar{H}$ of λ that has H as a subsequence. A locking λ of τ is called *(conflict-)safe*, if all histories of τ accepted by λ and projected on committed transactions only are *(conflict-)serializable*.

A locking λ obeys the *two-phase locking* policy [1, 10] if no transaction of λ is of the form where $Rl[x]$ or $Wl[x]$ for some x follows an unlock $Ru[y]$ or $Wu[y]$ for some y. A two-phase locking is *strict* [1] if all unlock instructions follow the abort or commit action in all transactions.

Our claim is that strict two-phase locking is unnecessarily restrictive because locks can be released only after the transaction's commit has been processed. If the group commit [6] optimization is applied, this would mean quite long delays because in group commit several transactions' commits are grouped together in order to save disk writes. We shall show that locks can be released immediately after the commit request operation. This implies that group commits do not cause any extra delays in releasing locks compared with processing the commits one by one.

We say that a two-phase locking is *partially strict* if all unlock instructions follow the abort or commit request action in all transactions. Our goal here is to define such a locking which, in addition to the two-phase property, accepts partially strict histories. An immediate problem is the condition (iii) requiring that whenever ${c_i}^R$ precedes ${c_j}^R$ then also c_i precedes c_j. It is obvious that this requirement, called the *commit ordering*, cannot be achieved by using locks on data items.

Theorem 3. Whenever commit ordering is preserved, a partially strict two-phase locking accept only partially strict histories.

Proof. Let H be a history accepted by a partially strict two-phase locking λ. Then there is a legal history $\bar{H}$ of λ that has H as a subsequence. Assume that H is of the form

$$\ldots W_j[x] \ldots O_i[x] \ldots,$$

where $O_i[x]$ denotes either $R_i[x]$ or $W_i[x]$. Because λ is a two-phase locking and $\bar{H}$ is legal, we conclude that $\bar{H}$ is of the form

$$\ldots Wl_j[x] \ldots W_j[x] \ldots Wu_j[x] \ldots O_i[x] \ldots.$$

Partial strictness now means that ${c_j}^R$ or a_j precedes $O_i[x]$ implying that the first two conditions of partial strictness are fulfilled. By the assumption that commit ordering is preserved, we conclude that H is partially strict. □

Our next question is how commit ordering can be implemented. We first look at the possibility of using locks. We may introduce into a locking new lock and unlock instructions that are not identified with the data items but with the c^R or c actions or with some specific locking variables (see, e.g., [10]).

Let $\tau = \{T_1, \ldots, T_n\}$ be a set of transactions, and let λ be its locking. Here we assume that, in addition to the lock and unlock instructions identified with the data items, a locked transaction may contain steps $Wl[x]$ and $Wu[x]$ where x is not a data item. The requirements for a legal history apply also to these new $Wl[x]$ and $Wu[x]$ steps but the definitions for a two-phase locking and strict and partially strict two-phase locking only apply to lock and unlock steps identified with true data items. We say that λ is *commit locked* if for each transaction T of τ ending with c^Rc the transaction $\bar{T}$ is of the form

$$\ldots Wl[c^R]\alpha c^R Wl[c] Wu[c^R] c Wu[c],$$

where α does not contain any $R[x]$ or $W[x]$ action.

Theorem 4. A partially strict two-phase locking λ which is commit locked accepts only partially strict histories.

Proof. By Theorem 3 it is enough to show that λ preserves commit ordering. Let H be a history of the transaction set τ obtained from λ by deleting all lock and unlock steps, and assume that H is accepted by λ. For the sake of contradiction, assume that H does not preserve commit ordering, i.e., H is of the form

$$\ldots {c_i}^R \ldots {c_j}^R \ldots c_j \ldots c_i \ldots$$

for some transactions T_i and T_j of τ. There must be some history $\bar{H}$ of λ that is legal because λ accepts H. But this is clearly impossible because $Wl[{c_i}^R]$ precedes ${c_i}^R$ and $Wl[c_i]$ precedes c_i and $Wu[{c_i}^R]$. Thus we cannot set the lock $Wl[c_j]$ before the action c_i followed by $Wu[c_i]$. This means that no $\bar{H}$ is legal, and thus we conclude that H preserves commit ordering. □

In what extent have we realized commit ordering with commit locks? By Theorem 4 we know that the commit ordering is preserved by commit locks, but are there valid ordering that are forbidden by commit locks? Actually, using commit locks only two commit request operations may be present at a time without processing the next commit. That is, we are able to schedule partially strict histories

$$\ldots {c_{i1}}^R \ldots {c_{i2}}^R \ldots c_{i1} \ldots {c_{i3}}^R \ldots c_{i2} \ldots$$

but not, for example, histories

$$\ldots {c_{i1}}^R \ldots {c_{i2}}^R \ldots {c_{i3}}^R \ldots c_{i1} \ldots c_{i2} \ldots c_{i3} \ldots$$

Given a set $\tau = \{T_1, \ldots, T_n\}$ of n transactions we are able to construct a locking such that all valid orderings of commit request and commit actions will be realized. The question of how such lockings are constructed is discussed in detail in [7]. In our setting, however, the construction of such a locking is not feasible for two reasons. First, the number of locks needed is large ($O(n^2)$). Second, here we consider a dynamic situation in which we do not know in advance how many transactions will arrive.

In practice, however, it is easy to implement the commit ordering by collecting the transactions which have requested to be committed into a queue. The actual commits are then performed in this order. The group commit can be implemented such that always when a certain number of commit requests have arrived the corresponding commits will be done as a batch. Another possibility is to perform the group commit in certain time intervals when all thus far arrived and not yet done commit requests form the batch.

6 Conclusion

We have defined a new variation of the strict two-phase locking policy, called partially strict two-phase locking, that employs early releasing of locks in the sense that all locks of a transaction are released immediately after its commit request has arrived. We have proved formally that all accepted histories are recoverable and, whenever a transaction aborts without a preceding commit request, the recovery properties of strict two-phase locking are preserved. If a transaction must be aborted after its commit request (which can only happen in connection with rare system failures), these properties do not hold and we suggest that then all active transactions are aborted.

The initial motivation for this research was our analysis and implementation of shadow paging (see e.g. [1]) for managing transactions and recovery in

database systems [14]. Our shadow paging project was motivated by the need of creating database systems in which recovery would be extremely fast, without sacrificing much or anything of the desirable properties of a complete database management system. For efficiency reasons, in a system based on shadow paging concurrent transactions must be handled such that several transactions commit in a batch. In such an environment, it is important not to hold the transactions' locks until the whole batch has been processed.

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
2. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD*, pages 1–8, 1984.
3. M. H. Eich. A classification and comparison of main memory database recovery techniques. In *Data Engineering*, pages 332–339, 1987.
4. K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Treiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.
5. D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Engineering*, 4:63–70, 1985.
6. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
7. G. Lausen, E. Soisalon-Soininen, and P. Widmayer. On the power of safe locking. *Journal of Computer and System Sciences*, 40(2):269–288, 1990.
8. C.-C. Liu and T. Minoura. Effect of update merging on reliable storage performance. In *Data Engineering*, pages 208–213, 1986.
9. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.
10. C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.
11. Y. Raz. The principle of commitment ordering. In *Very Large Data Bases*, pages 292–312, 1992.
12. O. Wolfson. An algorithm for early unlocking of entities in database transactions. *Journal of Algorithms*, 7:146–156, 1986.
13. M. Yannakakis. A theory of safe locking policies in database systems. *Journal of the Association for Computer Machinery*, 29(3):718–740, 1982.
14. T. Ylönen. *Shadow Paging Is Feasible.* Licentiate's thesis, Department of Computer Science, Helsinki University of Technology, 1994.

This article was processed using the LaTeX macro package with LLNCS style