A Unified Approach to Concurrency Control and Transaction Recovery *
(Extended Abstract)

Gustavo Alonso[1], Radek Vingralek[2], Divyakant Agrawal[1], Yuri Breitbart[2], Amr El Abbadi[1], Hans Schek[3], Gerhard Weikum[3]

[1] Department of Computer Science,
University of California,
Santa Barbara, CA 93117
[2] Department of Computer Science,
University of Kentucky,
Lexington, KY 40506
[3] Department of Computer Science,
ETH Zurich
CH-8092 Zurich, Switzerland

1 Introduction

Conventional transaction management in shared databases is traditionally viewed as a combination of two orthogonal problems. One is to guarantee correctness when several transactions are executed concurrently. This is ensured by the concurrency control component of a transaction processing system. The other is to ensure the correctness of the database in spite of transaction aborts and system failures. This is ensured by the recovery mechanism. In both cases there are several criteria that impose increasing restrictions on executions, resulting in a hierarchy in which only the most restrictive criteria are chosen for implementing concurrency control and recovery mechanisms. The result is that the different criteria are incomparable except for the most restrictive criterion in each component. For instance, strict two phase locking and log based recovery with before images has become the de facto standard (i.e. rigorous schedules [BGRS91], once considered as a possible candidate for a unified treatment of concurrency control and transaction recovery).

Recently, Schek, Weikum and Ye [SWY93] have developed a unified theory for concurrency control and recovery. In this theory, the traditional concurrency control theory is expanded to include the actions of aborted transactions. As a result, the extended theory can be used to determine whether a given execution is correct both from a concurrency control as well as recovery points of view. A hierarchy of correctness classes is defined and the authors introduce a particular class, called *prefix-reducibility* (PRED). PRED is argued to be the most

* This material is based in part upon work supported by NSF grants IRI-9221947, IRI-9012902 and IRI-9117904 and by grant from Hewlett-Packard Corporation. This work has been performed while Y. Breitbart was on one year sabbatical and R. Vingralek was visiting for 8 months the database research group at ETH, Zurich.

permissive class of executions that are intuitively correct accounting for both concurrency and recovery in the system. The original paper left the construction of a scheduler based on PRED as an open problem.

In this paper, we analyze the characteristics of a scheduler to recognize the class of PRED executions. We start by proposing an equivalent definition for PRED. The original definition of PRED was stated in a non-procedural manner, which hinders its use for the development of dynamic schedulers. Our definition is procedural, and provides us with easy, graph testing protocols for recognizing all PRED executions. We also develop efficient schedulers based on a new class of locks.

Following a suggestion of the program committee of the EDBT'94 conference, this paper combines the results of [AAE93] and [VBSW93] in which closely related results have been developed independently and in parallel.

2 Model

A database is a collection of data objects. Users interact with the database by issuing transactions. A transaction is a partial order of read and write operations followed by either an abort or a commit operation. Two operations of different transactions conflict if they are performed on the same data object and one of them is a write. History H over a set of transactions $\mathcal{T}$ is a partial order of all the operations in the transactions from $\mathcal{T}$ such that this partial order must be in agreement with the partial order of each transaction from $\mathcal{T}$ and any two conflicting operations in H are ordered. In history H we also allow a set-oriented *group abort* operation $a(T_{i_1}, \ldots, T_{i_k})$, where $T_{i_1}, \ldots, T_{i_k}$ are from the transaction set $\mathcal{T}$. This operation indicates that an abort should be executed for each transaction from $T_{i_1}, \ldots, T_{i_k}$. However, the execution of these aborts is conducted concurrently.

A projection of a history H on a set of transactions $\mathcal{T}$ is a history that contains only operations of transactions from $\mathcal{T}$. A committed projection of history H is the history $C(H)$ that contains only operations of committed transactions in H. A *complete* history H over a set of transactions $\mathcal{T}$ is a history in which each transaction from $\mathcal{T}$ is either committed or aborted. In what follows we will use the standard notions of serializability, recoverability, avoidance of cascading aborts, and strictness [BHG87].

In order to handle aborted transactions explicitly in the history we replace a transaction abort statement with a sequence of the transaction's undo operations. Each operation of an aborted transaction T_i that changes the database state, i.e. a write operation in our model, is replaced in the history with its inverse operation denoted by o_i^{-1}. On the other hand, the operations of T_i that do not impact the database state, in our case read operations, can be safely discarded from the history since the returned values are rendered invalid by the abort. Thus, if the scheduler produces a serializable history of transactions operations for committed transactions and undo operations for aborted transactions by discarding read operations of aborted transactions, then issues of serializabil-

ity and recovery are treated by such a scheduler in a uniform way. To guarantee correct recovery, it is assumed that active transactions (i.e., transactions that neither commit nor abort in H) will abort. These ideas proposed in [SWY93] lead to the following definition of an *expanded history*:

Definition 1. Expanded Histories

Let $H = (A, <)$ be a history. Its expansion $\tilde{H}$, or expanded history $\tilde{H}$, is a tuple $(\tilde{A}, \tilde{<})$ where:

1. $\tilde{A}$ is a set of actions which is derived from A in the following way:
   (a) For each transaction $T_i \in H$, if $o_i \in T_i$ and o_i is not an abort operation, then $o_i \in \tilde{H}$
   (b) Active transactions are treated as aborted transactions, by adding a set-oriented abort $a(T_{i_1} \ldots T_{i_k})$ at the end of H, where $T_{i_1} \ldots T_{i_k}$ are all active transactions in H.
   (c) For each aborted transaction $T_j \in H$ and for every write operation $w_j(x) \in T_j$, there exists an inverse write $w_j^{-1}(x) \in \tilde{H}$, which is used to undo the effects of the corresponding write operation. An abort operation $a_j \in H$ is changed to $c_j \in \tilde{H}$.
2. The partial order, $\tilde{<}$, is determined as follows:
   (a) For every two operations, o_i and o_j, if $o_i < o_j$ in H then $o_i \tilde{<} o_j$ in $\tilde{H}$.
   (b) If $a(T_{i_1}, \ldots, T_{i_k}) \in H$, then every two conflicting undo operations of transactions from the set $\{T_{i_1}, \ldots, T_{i_k}\}$ are in $\tilde{H}$ in a reverse order of the two corresponding write operations in H.
   (c) All undo operations of every transaction T_i that does not commit in H follow the T_i original actions and must precede c_i in $\tilde{H}$.
   (d) Whenever $o_n < a(T_{i_1}, \ldots, T_{i_k}) < o_m$ and some undo operation o_j^{-1} ($j \in \{i_1, \ldots, i_k\}$) conflicts with o_m (o_n), then it must be true that $o_j^{-1} \tilde{<} o_m$ ($o_n \tilde{<} o_j^{-1}$).
   (e) Whenever $a(\ldots, T_i, \ldots) < a(\ldots, T_j, \ldots)$ for some $i \neq j$, then for all conflicting undo operations of T_i and T_j, o_i^{-1} and o_j^{-1}, it must be true that $o_i^{-1} \tilde{<} o_j^{-1}$.

Note that this definition of expanded histories is more restrictive than that in [SWY93]. With the original definition, the expansion of the history $w_1[x]w_2[x]a_1a_2$ could be either $w_1[x]w_2[x]w_1^{-1}[x]c_1w_2^{-1}[x]c_2$ or $w_1[x]w_2[x]w_2^{-1}[x]c_2w_1^{-1}[x]c_1$. The intuitively correct expansion is the first one and so is assumed in [SWY93]. However, the definition provided did not rule out the second expansion, which is undesirable because it delays the abort operation issued by T_1.

A history H is *reducible* (RED) if its expansion $\tilde{H}$ can be transformed to a serial history by applying the following three transformation rules finitely many times.

1. **Commutativity Rule.** If two operations o_i and o_j do not conflict, and there is no o_k such that $o_i \tilde{<} o_k \tilde{<} o_j$, then the ordering $o_i \tilde{<} o_j$ can be replaced by $o_j \tilde{<} o_i$

2. **Undo Rule.** If $w_i[x]$ and $w_i^{-1}[x]$ are in $\tilde{H}$ and $w_i[x] \tilde{<} w_i^{-1}[x]$ and there is no o_j such that $w_i[x] \tilde{<} o_j \tilde{<} w_i^{-1}[x]$, then $w_i[x]$ and $w_i^{-1}[x]$ can be omitted from the expanded history
3. **Null Action Rule.** Read operations of aborted or active transactions in H can be omitted from $\tilde{H}$.

Unfortunately, the class RED is not prefix-closed and hence cannot be used for on line scheduling of transactions. This is resolved by further restricting this class of histories. In particular, a history H is *prefix reducible* (PRED) if every prefix of H is reducible. If we use the term SR-RC for the class of histories that are both serializable and recoverable, it can be shown that PRED $\subset$ SR-RC. Similarly, if by SR-ST we denote the class of histories that are both serializable and strict, SR-ST $\subset$ PRED [SWY93].

3 Serializability with Ordered Termination

We start this section by describing in more detail the standard recovery mechanism [HR83, BHG87, GR93], which is also assumed in the model proposed in [SWY93]. In this model, a *log* is maintained on stable storage for recovery purposes. Note that virtually all commercial systems employ a log-based recovery method [GR93], whereas deferred update methods have not been adopted in practice for performance reasons. The log is composed of a set of log records, each corresponding to a write operation. The log is used for two main purposes. First, when a transaction aborts, it is used to retrieve the transaction's log records for undoing the updates of the transaction. Second, after recovery from failures, the log is traversed by a special restart routine that redoes missing updates of all committed transactions and undoes the updates of all aborted or active transactions. A common implementation technique for transaction undo and redo is the restoration of before-images and after-images of the database objects; however, undo and redo may also be based on more general operations at the storage level.

We now consider the various restrictions that are imposed by log-based recovery on the order in which transactions may terminate. In particular, we are interested in the restrictions, if any, imposed on the commitment or abortion of transactions when they execute conflicting operations, i.e., wr, ww, and rw conflicts. These restrictions are as follows. To guarantee recoverable histories, if $w_1[x] < r_2[x]$ then $c_1 < c_2$. Transaction undo involves the invocation of undo steps that reinstall the state of database objects as of the time before the update that is undone; the order of the undo steps is the reverse of the order of the corresponding forward operations. Thus, if $w_1[x] < w_2[x]$ then if both transactions abort, $a_2 < a_1$ or $a(T_1, T_2)$; if T_1 aborts then T_2 cannot commit; and if T_1 commits and T_2 aborts, then there is no restriction on the termination order. Finally, the fact that commit decisions must be made on-line without knowing the future outcome of active transactions forces that if $w_1[x] < w_2[x]$ then $c_1 < c_2$; otherwise, if T_2 committed first, then a subsequent abort of T_1 could not be realized

properly by means of log-based undo steps. These restrictions are formalized in the following definition.

Definition 2. Serializability with Ordered Termination (SOT)
A history H is SOT if it is recoverable, serializable, and for every pair of conflicting operations w_i and w_j in H such that w_i precedes w_j and a_i cannot appear before w_j, the following conditions hold:

1. if T_j commits then it commits after T_i commits and
2. If T_i aborts then either it aborts after T_j aborts or H contains a group abort $a(\ldots T_i \ldots T_j \ldots)$.

Note that in the definition of SOT, serializability is with respect to the committed projection of the history, while the constraints on the termination of transactions are in terms of both committed as well as aborted transactions in the history. In [AVA+94] we prove the following theorem.

Theorem 3. *The class of histories SOT is equivalent to the class of histories PRED.*

Although intuitively PRED is the largest class of executions that is correct from both the concurrency control and the recovery points of view [SWY93], PRED executions may result in cascading aborts. Traditionally, cascading aborts occur when transactions are allowed to read uncommitted data. PRED executions may also result in cascading aborts when transactions write on uncommitted data. In particular, consider the following execution:

$$w_1[x]w_2[x]a_1$$

The abort of T_1 forces the abort of T_2 and a_2 must precede a_1 or they must be executed concurrently.

When restricted to the traditional methods of restoring before images, it can be shown that the class of strict and serializable histories is the maximal subclass of PRED that avoids cascading aborts [AVA+94]. To overcome this limitation, the next step is to modify the underlying recovery mechanism to expand the class of valid histories and still avoid cascading aborts. In [AVA+94] we extend log-based recovery to avoid cascading aborts due to write-write conflicts and to allow transactions to abort regardless of the status of other transactions with which they may conflict.

4 Scheduling Protocols

The theorem in the previous section is fundamental since it provides a constructive characterization of PRED executions. This characterization serves to develop efficient protocols for enforcing PRED executions.

4.1 Serialization Graph Testing

Let H be a history. We expand the notion of the serialization graph $SG(H)$ of history H by considering direct conflicts between any two transactions. The edges of the graph are annotated with the type of conflict between the two transactions (rw, wr, or ww). To guarantee SOT histories, the execution of transactions has the following three constraints: $SG(H)$ must be acyclic, if $T_i \rightarrow^{wr} T_j$ in $SG(H)$ then T_i must commit before T_j can commit, and if $T_i \rightarrow^{ww} T_j$ in H then either T_i commits before T_j terminates or T_j aborts before T_i terminates or both transactions abort concurrently.

These constraints imply that in case of a *wr* or *ww* conflicts there might be cascading aborts (note that the traditional notion of cascading aborts only applies to *wr* conflicts).

From the above constraints we derive the following serialization graph testing algorithm that recognizes SOT histories.

Let PRECEDE(T_j) be the set of transactions T_i such that $T_i \rightarrow^{wr/ww} T_j$. Let FOLLOW($T_j$) be the set of transactions T_k such that $T_j \rightarrow^{wr/ww} \ldots \rightarrow^{wr/ww} T_k$.

1. When an operation o_i that is not abort nor commit is submitted, add a node for T_i to the serialization graph, if it is not already there. Then add all corresponding *ww*, *wr*, and *rw* edges. If a cycle appears, a_i is submitted.
2. When c_i is submitted, the protocol checks whether PRECEDE(T_i) is empty. If this is not the case, c_i is delayed until all transactions in PRECEDE(T_i) have committed. Otherwise, c_i is executed. Once committed, T_i is removed from the graph and from all FOLLOW and PRECEDE lists.
3. When a_i is submitted, all transactions in FOLLOW(T_i) must also be aborted. If FOLLOW(T_i) is empty then a_i is executed. Otherwise, a(T_i FOLLOW(T_i)) is executed. This implies that undo operations to all write operations of all transactions in (T_i, FOLLOW(T_i)) are performed in reverse order than that in which they appear in the history . All aborted transactions are removed from the serialization graph, and from all immediate FOLLOW and PRECEDE lists.

Note that it is possible that an abort operation is issued when the transaction has already been aborted as a result of cascading aborts. In this case, the abort operation is acknowledged immediately without further action. We also assume that the scheduler has some mechanism ensuring that no two scheduled conflicting operations are processed at the same time.

The previous protocol is pessimistic. To reduce overhead, an optimistic protocol could be used. Its formulation is the same as above except that the graph is tested for cycles only when c_i is submitted. If there are no cycles, c_i gets processed in the same way as in the pessimistic protocol. Otherwise, c_i is rejected and a group abort is submitted that includes all transactions involved in a cycle. The detailed description and proofs of the correctness of these algorithms appear in [AVA+94].

4.2 Ordered Sharing Locking Protocol

Although the serialization graph testing approach recognizes, in general, all correct executions, it may incur unacceptable overhead. Many widely used concurrency control protocols use *locking* as a basic primitive for synchronization. Traditionally, there are two types of relationships between locks: shared and non-shared. For example, read locks can be shared but a write lock cannot be shared with any other lock. Two phase locking [BHG87] is the most widely accepted concurrency control protocol. Recently in [AE90], a new locking primitive was introduced that allows a new relationship, referred to as *ordered sharing*. Ordered sharing can be used with two phase locking to eliminate the blocking between read and write operations. For example, in order to eliminate read-write blocking, a transaction T_j can be granted a write lock on an object even if a transaction T_i holds a read lock on the same object. We say that there is an *ordered shared relationship* from T_i's read lock to T_j's write lock. The advantage of using ordered sharing is that it eliminates blocking between read and write operations, and it can be used to restrict the execution of commit or abort operations. To ensure SOT executions, the two phase locking with ordered sharing must observe the following rules:

1. *Lock Rule.* A transaction must obtain a read (write) lock on an object before executing a read (write) operation on that object.
2. *Two Phase Rule.* A transaction must not acquire any locks once it has released a single lock.
3. *Lock Acquisition Rule.* If T_j acquires a lock with an ordered shared relationship with respect to a lock held by another transaction T_i, the corresponding operation of T_j must be executed after that of T_i.
4. *Lock Relinquish Rule.* If transaction T_j acquired a lock with an ordered shared relationship with respect to a lock held by transaction T_i and T_i has not released *any* of its locks, then T_j cannot release any of its locks.
5. *Delayed commitment and abortion.* If T_j reads from T_i then T_j can commit only if T_i has committed and if T_j overwrites a value written by T_i then T_j does not commit before T_i terminates and T_i does not abort before T_j terminates.

The ordered sharing of locks is necessary because SOT does not impose any restriction on the order in which the operations are executed, as long as the overall history is serializable. With basic two phase locking, histories such as $w_1[x]w_2[x]w_1[y]w_2[y]$ are not possible, however they are correct and valid according to SOT. The rules of two-phase locking with ordered sharing guarantee the serializability of the committed projection of the history. The delayed commitment and abortion rule ensures that the expanded history is SOT. This protocol accepts a strict subset of SOT. However, we note that the SOT histories that are not accepted by this protocol are not order-preserving serializable histories[4].

[4] Order-preserving serializable histories maintain the order of non-interleaved transactions in the equivalent serial history [BSW79, BBG89].

5 Conclusions

In this paper, we have addressed an open problem posed by [SWY93]: how to characterize the class of histories PRED in a constructive way so that unified scheduling protocols can be derived from it. We have slightly modified the original definitions of expanded histories and PRED to account for certain executions, and we have provided an equivalent class, SOT, with a more constructive definition. This new class is used as the basis for several protocols that implement unified concurrency control and recovery in an efficient manner.

So far, our model is restricted to read and write operations. However, both the model and the developed protocols can be generalized to transactions with semantically rich operations where recovery is based on compensating operations.

References

[AE90] D. Agrawal and A. El Abbadi. Locks with Constrained Sharing. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 85–93, April 1990. To appear in Journal of Computer and System Sciences.

[AAE93] G. Alonso, D. Agrawal, A. El Abbadi. A Unified Implementation of Concurrency Control and Recovery. Technical Report, Department of Computer Science, University of California at Santa Barbara, TRCS93-19, October 1993

[AVA+94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H. Schek, and G. Weikum. Unifying concurrency control and recovery of transactions. *Information Systems*, 1994. to appear in the special EDBT'94 issue.

[BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewisz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transaction on Software Engineering*, 17(9), 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5(5):203–216, May 1979.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[HR83] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[SWY93] H. J. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 300–311, June 1993.

[VBSW93] R. Vingralek, Y. Breitbart, H.-J. Schek, G. Weikum. Concurrency Control Protocols Guaranteeing Atomicity and Serializability. Technical Report 199, Department of Computer Science, ETH Zurich, July 1993.