Dynamic Timestamps Ordering for Controlling Concurrency

A. (Zizo) Farrag

***Abstract*—This paper proposes several timestamp ordering mechanisms for controlling concurrency in which the timestamps assigned to transactions can be modified dynamically during execution. These timestamps are not stored with the (database) variables and the process of modifying them is simple. The proposed mechanisms achieve a higher level of concurrency (than traditional timestamp ordering mechanisms) for the following reasons. First, the operations of (certain classes of) read-only transactions can always be accepted. Second, when an operation by an update transaction arrives out of order, the mechanism avoids rejecting it by modifying, if possible, the timestamps of some transactions. The proposed mechanisms do not require multiversion of each entity to be maintained.**

***Index Terms*—Concurrency control, database, timestamps ordering, transaction.**

I. Introduction

In any multiuser database system, a concurrency control mechanism is needed to resolve any conflicts that might arise among transactions, and to ensure that their overall execution is correct (i.e., cannot violate consistency). Many such mechanisms have been previously proposed [1]-[13]; and most of them utilize some form of locking as a mean to control concurrency [4], [13]-[17]. That is, before a transaction can access any entity, it must first obtain an (appropriate) lock on it, and if this lock cannot be granted, the transaction will be delayed. This reduces concurrency and can potentially lead to deadlock. Resolving a deadlock requires aborting one or more transactions.

Other approaches to controlling concurrency avoid delaying the transactions by assigning each of them a unique timestamp, and then requiring all conflicting operations to be processed in a timestamp order (e.g., [9], [12], [14], [18], [19]). In this case, when an operation arrives out of order, it will be rejected, that is, the transaction that issued it will be aborted. Abortion is a serious drawback, and can degrade performance if it occurs frequently.

To avoid unnecessary abortions, we propose a dynamic timestamps ordering method which allows the timestamps of transactions to be modified during execution. The timestamps are not stored with the entities in the proposed method, and the process of modifying them is a simple operation. As will be shown later, separating timestamps from entities will help to solve issues that arise in other timestamp ordering mechanisms. The proposed mechanisms do not require multi-version of each entity to be stored, but it maintains a digraph that represents the "conflicts" among transactions.

The complexity is reduced by searching the digraph only when an operation arrives out of order; otherwise, the operation will be accepted without search.

The rest of the paper is organized as follows. The formulation used throughout the paper is given in Section II. The proposed mechanism is presented in Section III. Generalizing this mechanism is examined in Section IV. Section V presents conclusions.

Manuscript received June 5, 2015; revised November 14, 2015.

A. (Zizo) Farrag is with Dalhousie University, Canada (e-mail: farrag@cs.dal.ca).

II. Formulation

The database is modeled as a collection of entities or variables denoted $\{x, y, \ldots, z\}$ and each transaction is modeled as a sequence of read or write operations (O_1, O_2..., etc.) on these variables. A read and a write operation by transaction T_i on a variable x will be denoted $R_i(x)$ and $W_i(x)$; respectively. When this read (or write) operation is accepted, it returns (or modifies; respectively) the current value of x. A transaction is assumed to represent a correct computation, i.e., it can only be committed after all its operations have been accepted. Each new transaction will be assigned a unique name such as T_1, T_2, ..., etc, when it starts.

Transactions are classified into two types, read-only (R) and update (U) transactions. A transaction of type R does not modify any database variables, whereas a transaction of type U modifies at least one variable. The type of transaction T_i is denoted $TY(T_i)$.

Each update transaction T_i will be assigned a unique timestamp, i.e., a number denoted $TS(T_i)$. The timestamp of T_i does not have to be fixed during its execution, that is, may change from time to time by the concurrency control (to avoid backing-up T_i). Timestamps are used to synchronize "conflicts" among transactions as will be explained shortly. Although timestamps are allowed to change, the current timestamps of all transactions must be distinct. No timestamp is required for a transaction of type R. (This assumption will be relaxed later when we generalize type R transactions.)

For now we assume timestamps are assigned using a counter named TS-count initialized originally as 1. When a new update transaction starts, its timestamp will be calculated as the current value of TS-Count, and then this value will then be incremented by 1.

A. Definition II.1 (Schedule)

An interleaved sequence of the operations of a set of transactions is called a schedule. Consider, for example, the following three transactions:

$$T_1 = R_1(x)W_1(x)R_1(y)$$

$$T_2 = W_2(x)W_2(y)$$

$$T_3 = R_3(x)$$

An example of a schedule S of $\{T_1, T_2, T_3\}$ is $S=R_1(x)R_3(x)W_1(x)W_2(x)R_1(y)W_2(y)$.

The order of operations in the schedule (and in the transactions) increases from left to right. The order of operations of the same transaction must be preserved in S. A read operation $R_i(x)$ returns the value of x written by $W_j(x)$, if $W_j(x)$ precedes $R_i(x)$ in the schedule and no other write operation on x appears between $R_i(x)$ and $W_j(x)$. If no write operation on x precedes $R_i(x)$, $R_i(x)$ will return the initial value of x (that existed in the database before executing the schedule).

When the operations of each transaction appear in the schedule consecutively (i.e., without interleaving), the schedule is said to be serial.

B. Definition II.2 (Conflicting Operations)

Two operations belonging to two different transactions are said to be conflicting operations iff both operations access the same variable and at least one of them is a write operation.

C. Definition II.3 (Schedule Equivalence)

Two schedules S_1 and S_2 for the same set of transactions are said to be equivalent iff for each pair of conflicting operations O_i and O_j, these operations must have the same relative order in S_1 and S_2; i.e., O_i precedes O_j in S_1 iff O_i precedes O_j in S_2.

The above definition guarantees that each transaction sees the same database in both schedules. For example, it is not hard to see that the following two schedules are equivalent

$$S_1 = W_2(x)W_1(x)R_3(x)R_1(z)W_2(y)R_3(y)R_3(z)R_2(z)W_4(z)$$

$$S_2 = W_2(x)W_2(y)R_2(z)W_1(x)R_1(z)R_3(x)R_3(y)R_3(z)W_4(z).$$

D. Definition II.4 (Serializable Schedule)

A schedule S is said to be serializable iff it is equivalent to a serial schedule. For instance, the schedule S_1 in the above example is serializable since it is equivalent to the given serial schedule S_2.

E. Definition II.5 (Dependency Digraph)

For any schedule S, the dependency digraph of S, denoted DD(S), is a directed graph whose nodes correspond to the set of transactions in S, and contains an arc (T_i, T_j) iff there is an operation by T_i in S which precedes and conflicts with an operation by T_j.

To prove the correctness of a concurrency control algorithm, it suffices to show that for each schedule S produced by it, DD(S) is acyclic or that S is equivalent to a serial schedule of the same set of transactions.

III. Dynamic Timestamps Ordering

This section presents a dynamic timestamp ordering mechanism in which the timestamps previously assigned to transactions can be changed during execution. In this mechanism, the operations of a read-only transaction will always be accepted. The mechanism maintains a dependency digraph (according to the rules defined earlier). This digraph is not used to check for a cycle, since it is always kept acyclic; rather, it is used for re-modifying the timestamps of transactions to avoid rejecting an operation (as will be shown later). A generalization of this mechanism will be discussed in the next section.

A. Values Maintained

The mechanism uses the following values in processing the operations.

TR(x) and *TW*(x)*:* These denote the names of the transactions with the largest timestamps that has read and written x; respectively. The name will be recorded when the operation is accepted. These values may not necessarily be for currently running transactions. Initially, before x is read or written by any transaction, *TR*(x) and *TW*(x) denote a special transaction named T_0 whose timestamp is set as 0 and will never be modified by the mechanism. Further, T_0 will not be represented by a node in the dependency graph.

Flag(x): This flag is used to accept the operations by read-only transactions on x. It is initialized as 0, will be incremented by 1 when a read-only transaction T_i reads x, and will be decremented by 1 when T_i is terminated.

Active-Set: This contains the names of all currently active transactions. When a new transaction starts, its name will be added to the set and when it is terminated, its name will be removed.

TS(T_i): The mechanism maintains the timestamp *TS*(T_i) for each update transaction T_i that is currently active, or its name is still recorded in the variable *TR*(x) or *TW*(x) (Notice timestamps are not stored with the database variables). Timestamps are generated using the counter TS-Count as described previously.

R-Set(T_i) and W-Set(T_i): These sets contain the database variables to be read and written by T_i; respectively. These sets are initialized as empty when T_i starts and will be removed when T_i is terminated.

TY(T_i): This denotes the type of each transaction T_i which is either R or U for read-only and update transactions; respectively.

B. Processing Operations

The procedures for processing operations and checking the dependency digraph are summarized below.

a) Procedure process-read

A read operation $R_i(x)$ will be accepted without searching the dependency digraph if T_i is a read-only transaction, or T_i is an update transaction and the condition $TS(TW(x)) < TS(T_i)$ is satisfied. Otherwise, the mechanism will first check the dependency digraph to see if timestamps can be modified in such a way that permits the acceptance of the operation, and if so, it will be accepted after modifying timestamps as described in Procedure Check_Digraph. (Notice after modifying these timestamps, the above condition will become satisfied.) Otherwise, if timestamps cannot be modified, the read operation will be rejected.

b) Procedure process-write

A write operation $W_i(x)$ will be accepted if $Flag(x) = 0$ and the condition $\text{Maximum}[TS(TR(x)), TS(TW(x))] \leq TS(T_i)$ is satisfied. However, if Flag(x)= 0 but the latter condition is not true, the mechanism will first check the dependency digraph

to see whether timestamps can be modified in such a way that permits the acceptance of the operation, and if so, $W_i(x)$ will be accepted after modifying timestamps as described in Procedure Check-digraph. Otherwise, $W_i(x)$ will be rejected (if Flag(x)! = 0; or $Flag(x) = 0$ but the timestamps cannot be modified).

c) Procedure check-digraph

The procedure to check the digraph (DD) to see if timestamps can be modified so as to avoid rejecting a read or write operation by a transaction T_i is as follows. First, we find the set of all transactions T_k reachable from T_i in DD. (This set will include T_i). If the operation is a read $R_i(x)$ and this set does not contain any transaction that wrote x, or if the operation is a write $W_i(x)$ and the set does not contain any other transaction that read or wrote x, the operation will be accepted after modifying timestamps as follows: for each transaction T_k in this set, its timestamp will be modified by adding TS-Count to its value, and following this we change TS-Count to become 2*TS-Count.

C. Committing Transactions

When a transaction T_i is committed, its updates will be become permanent, its node will be removed from the dependency digraph, and the user who submitted it will be notified of its successful completion. However, the time to commit a transaction depends primarily on the transaction model used. For now, we assume the same (general) model for a transaction defined in the preceding section (We shall discuss other models later [20]).

By the assumed model, a transaction T_i cannot be committed until the dependency digraph has no directed edges $T_j \rightarrow T_i$ as a result of T_i reading a value written by T_j. This condition implies that in any schedule S produced by the mechanism, if S has an operation by an update transaction T_u which precedes and conflicts with an operation by a read-only transaction T_r, then T_u was committed before T_r in S. Conversely, by (the conditions given in) the above procedures used for processing read and write operations, if the conflicting operation of T_r precedes that of the conflicting operation of T_u, this will imply T_r was committed prior to T_u. These two statements prove the following result.

Lemma III.1: Let S be a schedule produced by the mechanism, and let O_r and O_u denote two conflicting operations in S belonging to a read-only and update transactions T_r and T_u, respectively. Then, O_r precedes O_u in S iff T_r is committed before T_u.

D. Correctness of the Mechanism

Let S be a schedule of a set of transactions produced by the mechanism, we show below that the mechanism works correctly by proving S to be serializable. This also proves our earlier claim that the dependency digraph maintained will always be acyclic.

Theorem III.1:

Every schedule S produced by the mechanism is equivalent to a serial schedule H in which all update transactions are arranged according to their final timestamps, and a read-only transaction T_r precedes an update transaction T_u (in H) iff T_r is committed before T_u.

Proof: Let O_i and O_k denote two conflicting operations in S belonging to transactions T_i and T_k; respectively. Suppose first T_i and T_k are both update transactions. Then, since conflicting operations are processed in timestamps order, therefore, the order of these two conflicting operations in S must be the same as the order of (the final) timestamps of their corresponding transactions; and by the arrangement of transactions in H, this also implies that these two conflicting operations have the same order in H. Similarly, suppose instead T_i and T_k are update and read-only transactions; respectively. Then, by Lemma III.1, the order of committing these two transactions in S is the same as that of their conflicting operations; and since read-only and update transactions are arranged in H in the order they were committed, this in-turn implies O_i and O_k have the same order in both S and H.

Thus, every pair of conflicting operations have the same order in S and H, which means S is equivalent to H, i.e., S is serializable.

IV. Generalization

The proposed mechanism can be generalized in some useful ways as explained below.

A. Read-Only Transactions

The proposed mechanism has only one kind of read-only transactions with a higher precedence over update transactions, i.e., the former cannot be backed-up due to conflicts (with the latter). This can be generalized by allowing three kinds of read-only transactions as explained below.

Type R^1: This is identical to read-only transactions assumed before. That is, the operations of this transaction will always be accepted, and the values returned by these operations must satisfy any consistency constraints.

Type R^2: This has the same precedence as an update transaction (and lower precedence than that of R^1). That is, its operations will be processed in the same way as a read operation by an update transaction, and the values returned must also satisfy the consistency constraints of the database).

Type R^3: This has a lower precedence than the other 2 types of read-only transactions, since the values read by this transaction need not necessarily satisfy the database constraints; but its operations will always be accepted. (In other words, the operations of this transaction will not necessarily be serialized.)

To see the justification behind the above classification, consider a banking system in which the manager would like to know the total balance in all accounts combined. If an accurate amount is needed, the manager can run a transaction of type R^1 or R^2. However, since the balances of accounts change continuously, an absolute accuracy of the amount returned may not be essential and is not even guaranteed once the transaction is completed. In this case, the manager may run a transaction of type R^3 instead (to increase concurrency). Choosing between types R^1 or R^2 will depend on the transaction's priority and duration, i.e., R^1 can be used when the priority is high and execution time is relatively short.

B. Transaction Model

As mentioned earlier, the time to commit a transaction

depends primarily on the transaction-model used. Thus, for the (general) model defined before in Section II, a transaction T_i cannot be committed until every other transaction T_j that updated a (database) value read by T_i is committed first. Other restricted models might allow a transaction to be committed once its last operation is accepted as explained below.

One approach would be to group all write operations at the end of each transaction and to process them atomically, i.e., as an indivisible action. Another method, which is less restrictive, is to structure each update transaction as two phases [20], the first consists of all its read operations followed by the second phase that consists of all its write operations. With this model, we can prevent an update transaction say T_j from reading a value written by another update transaction say T_u until all write operations of T_u are processed successfully. This, however, would not prevent backing-up a read-only transaction (of type R^1 or R^2) as a result of backing-up an update transaction. However, backing-up a read-only transaction is simple, and cannot lead to backing-up any other transaction.

V. Conclusions

We have proposed several timestamp ordering mechanisms in which the timestamps of transactions can be modified dynamically during execution. In these mechanisms, the operations of (some classes of) read-only transactions can always be accepted, and moreover, when an operation by an update transaction arrives out of order, the mechanisms check first if timestamps can be modified in such a way that allows the acceptance of the operation. For these reasons, the mechanisms achieve a higher level of concurrency.

In comparison with other concurrency control algorithms that allow the operations of read-only transactions to be accepted, our mechanisms do not require multiversion of each database variable to be maintained (as in [2,12]), and the dependency digraph need not be searched every time an operation is processed (as in [2]).

References

[1] R. Agrawal and D. Dewitt, "Integrated concurrency control and recovery mechanisms: design and performance evaluation," *ACM Trans. Database Systems*, vol. 4, pp. 529-564, December 1985.

[2] R. Bayer, H. Heller, and A. Reiser, "Parallelism and recovery in database systems," *ACM Trans. Database Systems*, vol. 5, pp. 139-156, June 1980.

[3] N. Conway, W. Marczak, P. Alvaro, J. Hellerstein, and D. Maier, "Logic and lattices for distributed programming," in *Proc. the Third ACM Symposium on Cloud Computing*, San Jose, CA, USA, October 2012, pp. 1-10.

[4] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The notions of consistency and predicate locks in a database system," *Comm. ACM*, vol. 19, no. 11, pp. 624-633, November 1976.

[5] A. Farrag and T. Kameda, "On concurrency control using multiple versions," Technical Report TR 82-13, Dept. Computing Science, Simon Fraser University, BC, Canada, 1982.

[6] H. Garcia-Molina and G. Wiederhold, "Read-only transactions in distributed database," *ACM Trans. Database Systems*, vol. 7, pp. 209-234, June 1982.

[7] J. Gray, "Notes on database operating systems," *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 393-481, 1978.

[8] H. Kung and J. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Systems*, vol. 6, no. 2, Jun 1981, pp. 213-226, June 1981.

[9] L. Lamport, "Towards a theory of correctness of multi-user database Systems," Tech. Rep. CA 764-0712, Massachusetts Computer Associates, October 1976.

[10] F. Laux and T. Lessner, "Escrow serializability and reconciliation in mobile computing using semantic properties," *International Journal on Advances in Telecommunications*, vol. 2, no. 2&3, pp. 72-87, March 2009.

[11] T. Lessner, F. Laux, T. Connolly, and M. Crowe, "Transactional composition and concurrency control in disconnected computing," *International Journal on Advances in Software*, vol. 4, no. 3&4, pp. 442-460, 2011.

[12] D. Reed, "Naming and synchronization in decentralized computer systems," Tech. Rep.-205 Dept. Electrical Engineering and Computer Science, MIT, September 1978.

[13] D. Rosenkrantz, R. Stearns, and P. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Systems*, vol. 3, no. 2, pp. 178-198, June 1978.

[14] A. Farrag and M. Ozsu, "Towards a general concurrency control algorithm for database systems," *IEEE Trans. on Soft. Eng*, vol. SE-13, no. 10, pp. 1073-1079, October 1987.

[15] J. Gray, R. Lorie, G. Putzolu and L. Traiger, "Granularity of locks and degrees of consistency in a shared database," in *Proc. IFIP Working Conf. on Modeling of Database*, January 1976, pp. 695-723.

[16] Z. Kedem and A. Silberschatz, "Controlling concurrency using locking," in *Proc. the 20th International Conference on Foundation of Computer*, October 1979, pp. 274-285.

[17] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, "Adaptive locks: combining transactions and locks for efficient concurrency," *J. of Parallel and Distributed Computing*, vol. 70, no. 10, pp. 1009-1023, October 2010.

[18] R. Thomas, "A solution to the concurrency control problem for multiple copy databases," in *Proc. the Compcon Conference*, New York, USA, 1978.

[19] A. Umar and D. Teichroew, "Pragmatic issues in conversions of database applications," *Information & Management*, vol. 19, no. 3, pp. 149-166, October 1990.

[20] C. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631-653, October 1979.

A. (Zizo) Farrag is currently with the Faculty of Computer Science, at Dalhousie University, Halifax, NS, Canada, B3L4R2. His research areas include fault tolerance, theoretical computer science and databases. He has published in many important journals such as ACM-TODS, IEEE-TSE, IEEE-TPDS, networks, parallel computing (among others).