International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:1, 2008

A New Approach for Recoverable Timestamp Ordering Schedule

Hassan M. Najadat

***Abstract*—**A new approach for timestamp ordering problem in serializable schedules is presented. Since the number of users using databases is increasing rapidly, the accuracy and needing high throughput are main topics in database area. Strict 2PL does not allow all possible serializable schedules and so does not result high throughput. The main advantages of the approach are the ability to enforce the execution of transaction to be recoverable and the high achievable performance of concurrent execution in central databases. Comparing to Strict 2PL, the general structure of the algorithm is simple, free deadlock, and allows executing all possible serializable schedules which results high throughput. Various examples which include different orders of database operations are discussed.

***Keywords*—**Concurrency control, schedule, timestamp, transaction.

I. Introduction

THE number of users who access database applications has been growing so fast and it becomes the highest priority in database fields to handle transactions of users. Researchers have continuously made effort to enhance concurrency control and recovery for centralized database which stored at a single computer site. A centralized DBMS support multiple users, but the DBMS and the database themselves reside totally at a single computer site [10]. Centralized DBMS needs a concurrency control for transaction processing that coordinates the actions of processes that operate in parallel, access shared-data, and therefore potentially interfere with each other. Database products need to be accurate and recoverable. The work load of read and write operation requests the systems to execute a large number of transaction at a given time.

In [2], many existing database applications place various timestamps on their data. Timestamp is a unique identifier created by the DBMS to identify a transaction [6]. ts(Ti) refers to transaction with assigned a timestamp i. The timestamp of any transaction is represented by its index (i.e., index(Ti) = i). The index of all operations in Ti is i. Each data item contains a read timestamp, giving the timestamp of the last transactions to read the item and a write timestamp, giving the timestamp of the last transaction to write the item [10]. Timestamp ordering operations are conflict if they come from different transactions; if they operate on the same data item and either at least one of them is a write operation. The timestamp ordering rule states that if pi(x) and qj(x) are conflicting operations then pi(x) is processed before qj(x) if and only if ts(Ti) < ts(Tj).

The timestamp ordering protocol is used to ensure serializability based on the order of transaction timestamps [4]. The basic timestamp ordering enforces conflict serializability but it does not ensure recoverable schedules; and hence it does not ensure cascadless or strict schedules either [3]. The definition of recovery, avoid cascade, and strict schedule is as follows:

In [11], every history H consists of a set of transactions with two parts: a set of events that reflects the operations (e.g., read, write, abort, commit), and a version order in chronological execution.

Recovery schedule: A history H is called recoverable (RC) if, whenever Ti reads from Tj ($i \neq j$) in H and $Ci \in H$, Cj < Ci. A history is recoverable if each transaction commits after commitment of all transactions (other than itself) from which it read [6]. The basic timestamp ordering enforces conflict serializability but it does not ensure recoverable schedules. The recoverable systems increase throughput that affects the speed processing. The present algorithm guarantees to produce a recoverable schedule without delaying the processing data.

Avoid cascade schedule: If, whenever Ti reads x from Tj ($i \neq j$) in H and Ci < Ri[x], it avoids cascading aborts (ACA). That is, a transaction may read only those values that are written by committed transactions or by it.

Strict schedule: the schedule is strict (ST) if, whenever Wj[x]<Oi[x] ($i \neq j$), either abort j < Oi[x] or Cj < Oi[x] where Oi[x] is Ri[x] or Wj[x].

In this work, a new algorithm for recoverable time stamping is provided with a new definition of recoverability property and premature commit. We provided a formal proof of our algorithm, and then we applied this algorithm in many models. This paper is organized as follows. Section II provides an abstract model for our approach with a discussion of the definition of recoverable property. Section II defines the transaction and recoverable history mathematically. Section III presents a detailed discussion of the new algorithm. We conclude with section V.

Hassan M. Najadat is currently assistance professor at Computer Information Systems Department in Jordan University of Science and Technology, P.O. Box 3030 Irbid 22110 Jordan (phone: 962-2-7201000- ext. 23405; fax: 962-2-7095123; e-mail: najadat@ just.edu.jo).

II. Database Model

The components of database system as shown in Fig. 1 include: (1) transaction manager, which performs any required preprocessing of database and transaction operations it receives from transactions, (2) a scheduler, which controls the relative order in which database and transaction operation are executed, (3) a recovery manager, which is reasonable for transaction commitment and abortion; and (4) a cache manager, which operates directly on the database [10]. Most of the concurrency control techniques ensure serializability of schedules. Using timestamps to order transaction execution guarantees serializability.

The most important operations considered in this work include: *Ri[x]* reads database item named x, where *i* is the transaction index, *Wi[x]* writes the value program variable x into the database item named x, where *i* is the transaction index, and C*i* tells the DBS that the transaction i has terminated normally and all of its effects should be made permanent.

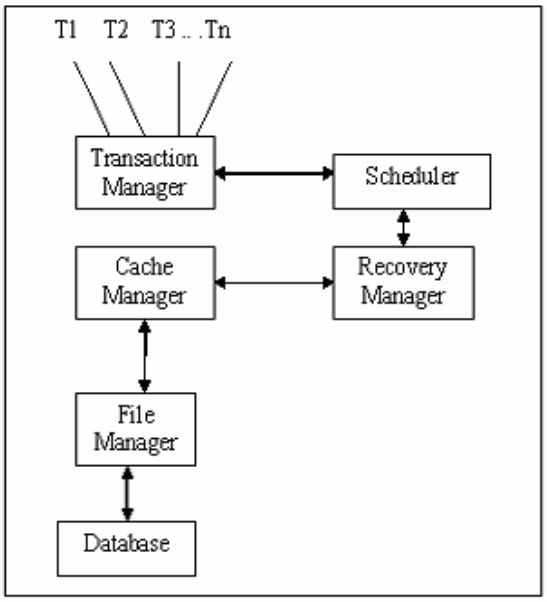

Fig. 1 Database system components

A history model composed of Reads and/or Writes, and their execution order. We used this model to express the execution of a set of transactions. There is more detail in [1].

The recoverability property ensures that a history is recoverable if each transaction commits after the commitment of all transactions (other than itself) from which it reads. This section defines formally the recoverable property. Active/abort transactions are not considered and the factor of pre-existing database prior to an execution is not included in the history model.

Two operations are *conflict* if they come from different transactions; if they operate on the same data item and either at least one of them is a write. A new concept called a *premature commit* is defined as follows: Wi [x]@t is the most recent *write* of x for Rj[x] @ t' such that t < t', Cj is called a *premature commit* if and only if Cj < Ci. A history ensures the *recoverability property* iff it has no Premature Commit.

A *recoverable history***,** H, ensures the recoverability property for each set of chronological operations. The following histories demonstrate the recoverability property of H1, H2, and H3.

If H1 is defined as *H1: W2(x) R2(y) C2 R1(x) C1* then H1 is considered to be recoverable because T2 writes x, and then commits before T1 reads x. H2 is also considered recoverable as sequence of the following operations: *H2: W1(x) R2(x) R1(y) W2(x) W1(y) R2 (z) W2 (z) C1 C2*. H3 represents a non-recoverable history as follows *W1(x) R2(x) W2(x) R2(y) W2(y) C2 R1(y) W1(y) C1*. In H3, *C2* is a premature commit; it happened before C1. Our algorithm enforces H3 to be recoverable without changing the chronological execution of operations except the commit operation by delaying C2 until C1 commits. In the following section, we give a description of the new approach which remedies this problem.

III. New-RCTO Algorithm

In this work, the following assumptions are considered: (1) the proposed approach follows basic timestamp ordering except in commit operation, and (2) for every data item x, there are two vectors: (i) the write vector, *wv*, records the write timestamp for each write/read operation, and (ii) the commit vector, *cv***,** records commit operation for each transaction only if the timestamp of received commit operation does not equal the timestamp of the first stored element in wv.

A rotate function shifts the elements of *wv* such that the first element in wv shifts to the last location and the remaining elements move up as in Fig. 2.

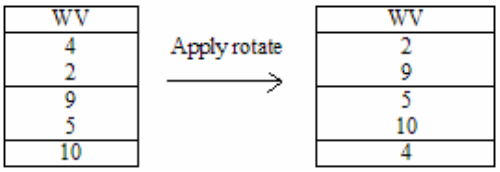

Fig. 2 Rotate function example

NEW-RCTO algorithm consists of three different phases: (1) *read phase*, (2) *write phase*, and (3) *commit phase*. Before defining these phases, the timestamp of commit operation for different transaction on the same object maintains a relationship as depicted in the following code:

```
1. if ts(Ci) = ts(wv[i])
   {
2.   Execute Ci,
3.   Delete the contents of wv[i],
4.   Move up the remaining values of wv
   }
```

The above code operates simultaneously in both wv and cv

vectors in order to get a high speed in processing of commit operation.

Fig. 3 depicts a schedule execution which consists of the following operations: C3C4C5 is a set of three commit transactions, t1,t2,and t3, stored in commit vector, while W2(x) W4(x) W1(x) is a set of write operation on the same object x stored in write vector for the three transactions. Since C4 and W4 are stored in the same row with the same index, they will be deleted from wv and cv simultaneously. The remaining w2, c3, w1 and c5 operations will be moved up one row.

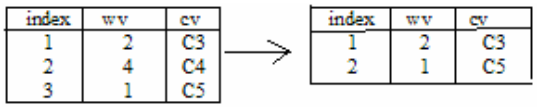

Fig. 3 Write vector and commit vector status after matching w4 c4

A. Read Phase

Whenever read operation, Ri, arrives, the following code will be executed:

```
1.  For each received read operation i
    {
2.  Let z = i * random number
3.  If ts(Ri) is already in wv
     then
4.        store z instead of Ri;
     else
5.        store z at the end of wv
    }
```

Fig. 4 depicts the *read phase* with the last data stored in Fig. 3. Suppose the next arrival operation is R2(X), then apply read phase will result the right table. Since the timestamp of read operation is already stored in wv[2], the contents of wv[2] is multiplied by a random value. The main advantage of this phase is to distinguish the read operation from the write operation which will not cause any delay in execution of the read operation.

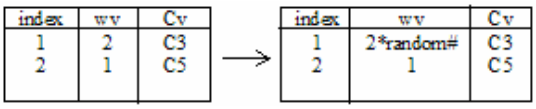

Fig. 4 Read operation R2 arrived and then multiplied by random value

B. Write Phase

Whenever write operation, Wi, arrives, the following code will be executed:

```
1.  For each received write operation i
{
2.  If ts(Wi) is not stored in wv then
3.  Save  ts(Wi) in wv
}
```

The write step enforces the write operation to store its timestamp in wv. Fig. 5 shows the contents of wv and cv after a new operation arrived, i.e. W3(x). The initial data stored is shown in the left side of Fig. 5 and the right side is the result after W3 arrived.

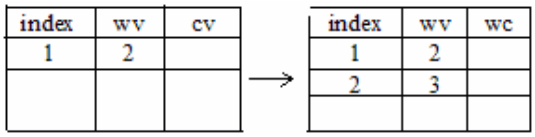

Fig. 5 Write vector and commit vector status after executing W3

C. Commit Phase

This phase is the main phase that enforces all commit operation either to compare the ts(Ci) with the first element in wv or to keep commit operation in cv. The following code represents the commit phase:

```
1.  Let k is the index of the first empty available position
    in cv.
2.  For each received Commit operation Ci
    {
3.     If (ts(Ci) = wv[0]) or (ts(Ci) =wv[k])
          {
4.          Execute Ci;
5.          Delete wv[0]  Or Delete wv[k];
6.           Move up all the remaining values in wv and cv
             simultaneously
          }
    else
       {
7.           Record Ci at the first  empty cell in cv
       }
    }
```

The main advantage of commit phase is to delay any premature commit and enforce the commit of transaction of write operation to be executed before any commit of different transaction for read operation. Fig. 6 shows the commit phase after C3 arrived. Since W3 is already stored in wv, the commit operation executes and delete wv[2] contents -i.e.W3.

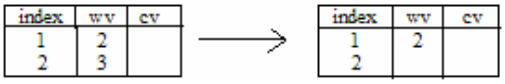

Fig. 6 C3 commit arrived and then delete W3 from wv[2]

D. Model of new-RCTO

Two histories of execution are provided to complete our discussion of new-RCTO by combining all phases. The first represents the historical execution order H for all operations in transaction 1 and transaction 2, ts(Ti)=i, index(Ti)=i, and index any operation in Ti equal i.

Let a history H1= W1(x) W2(x) W3(x) R4(x) C1 C4 C3 C2. we can observe the following:

a) W1(x) arrives, New-RCTO dispatches ts(W1(x)) to

wv[1].

b) W2(x) arrives, RCTO dispatches ts(W2(x)) that equal 2 to wv[2].

c) W3(x) arrives, ts(W3(x)) dispatches ts(W3(x)) that equal 3 to wv[3].

d) R4(x) arrives, RCTO multiplies ts(R4(x)) by a random number and compares it to the contents of wv, if the matching successes then RCTO doesn't add it else RCTO add it. The comparing results fail and 4 *rand recorded to wv[3]. Fig. 7 depicts a, b, c and d.

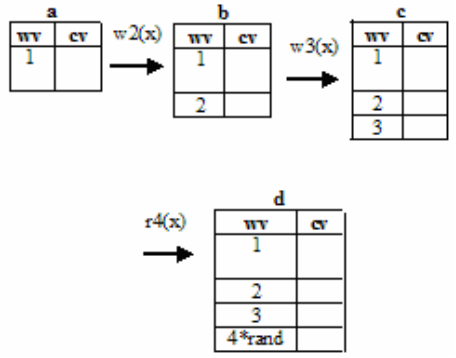

Fig. 7 H=W1(x)W2(x)W3(x)R4(x)C1C4C3C2

e) C1 arrives, RCTO checks if ts(C1) = wv[1], the comparing success, New-RCTO execute C1 and delete cv[1]. All remaining values in wv move up on location.

f) C4 arrives, the comparing ts(C4) with contents of wv[1] fails, New-RCTO records C4 in first location in cv.

g) C3 arrives , New-RCTO checks if ts(C3) = wv[1],it fails, the recording C3 will be in cv[2], but the value in wv[2]=ts(cv[2]), then execute C3 and delete wv[2], move up all elements in wv on location. Fig. 8 shows e, f, and g.

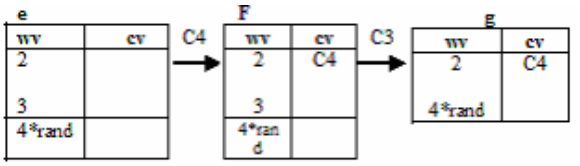

Fig. 8 H=W1(x)W2(x)W3(x)R4(x)C1C4C3C2

h) C2 arrives and compares with wv[1] success. new-RCTO executes C2 and deletes wv[1], and also the remaining elements

i) There is only one value in wv and C4 in CV, then execute C4 and delete 4*rand. Fig. 9 shows h and i.

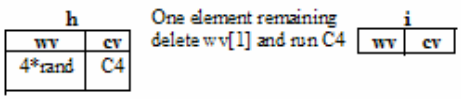

Fig. 9 H=W1(x)W2(x)W3(x)R4(x)C1C4C3C2

E. Correctness of New-RCTO

The most powerful of NEW_RCTO is producing much high system throughput with non-deadlock happened. To proof the correct of the proposed approach, we introduce a formal proof as follows:

Theorem: If H is a history produced by New-RCTO then H is recoverable.

Proof:

We use the serialization graph as a tool to proof the above theorem. Serialization graph SG used to test a schedule for conflict serializability. It looks only at read/write operations in the schedule to construct a serialization graph.

The serialization graph is a directed graph. There is one node for each transaction and a set of directed edges, each edge in the graph is of the form (Ti→ Tj) where Ti is starting node and Tj is the ending node. E is constructed to the following rule: If R(x) @t reads from W[x] @ t', add R[x]t, W[x]t' to the graph. This rule states that, for each reads-from relationship, an edge is added to the graph. Now we can proof the theorem:

Prove by contradiction
Let H be a history whose SG is not recoverable
The assumption that H is not recoverable means that there is a premature commit
Let R(x) @t' reads from the most recent W(x) @ t and Ct'< Ct
This implies
W(x) @ t < R(x) @t'
Ct'< Ct
t' > t.
Clearly,
The execution Ct'< Ct does not be allowed in new-RCTO.
new-RCTO delayed Ct' until executes Ct
This execution guarantees Ct < Ct'
This result conflicts with the above assumption therefore H is recoverable.

IV. Conclusion

This paper has incorporated new-RCTO into concurrency control in centralized database. This offered a recoverable execution of transactions. As shown in different applications, new-RCTO guaranteed the execution output by the scheduler to the data manager to be recoverable. We provided a comprehensive model to complete understanding of new-RCTO. All write/read operations are executed without any delay.

References

[1] P. A. Bernstein, V. Hadzilacos, N. Good-Man, "Concurrency control and recovery in database systems", Addison Wesley, 1987.
[2] T. Kristian, S. J. Christian, T. S. Richard, "Effective time stamping in databases, a time center", Technical report, 1998.

[3] Elmasri, Navathe, "Fundamentals of database systems", Addison Wesley, 2nd edition 1994.
[4] Ramon Lawrence, "Advanced database seminar-concurrency control", Lecture notes, 2001
[5] D. Jeffrey, Ullman, "Principles of database and knowledge-base systems", W.H. Freeman & Company, 1988.
[6] A. James, A. Fekete, N. Lynch, M. Merrittt, W. Weihl, "A theory of timestamp-based concurrency control for nested transactions", Proceedings of the 14th VLDB Conference, 1988.
[7] J. T. Young, J. C. Peck, "A mathematical theory of correct executions in temporal databases supporting concurrent simulations", ACM ,1998
[8] W. Smith, D. B. Johnson, "Minimizing timestamp size for completely asynchronous optimistic recovery with minimal rollback", 15th IEEE Symposium on Reliable Distributed Systems, 1996.
[9] T. Connolly, C. Begg, A. Strachan, "Database systems, A practical approach to design, implementation, and management", Addison Wesley, 4nd edition, 2004.
[10] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions", proceedings of the IEEE International Conference on Data Engineering, san Diego, CA, March 2000.
[11] M. Lonescu, B. Dorohonceanu, I. Marsic," A novel concurrency control algorithm in distributed groupware", Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA '2000), p. 1551-1557, Las Vegas, NV, June, 2000.