DEADLOCK PREVENTION IN A DISTRIBUTED DATABASE SYSTEM

P.KRISHNA REDDY and SUBHASH BHALLA

School of Computer and Systems Sciences
Jawaharlal Nehru University
New Delhi-110 067(INDIA)
(E-mail: bhalla@jnuniv.ernet.in)

ABSTRACT

The distributed locking based approaches to concurrency control in a distributed database system, are prone to occurrence of deadlocks. An algorithm for deadlock prevention has been considered in this proposal. In this algorithm, a transaction is executed by forming wait for relations with other conflicting transactions. The technique for generation of this kind of precedence graph for transaction execution is analyzed. This approach is a fully distributed approach. The technique is free from deadlocks, avoids resubmission of transactions, and hence reduces processing delays within the distributed environment.

1.0 INTRODUCTION

The deadlock problem is intrinsic to a distributed database system, which employs locking as a means of supporting concurrency control algorithm. A deadlock occurs when a transaction, waits for locks held by another transaction which is also waiting (directly or indirectly) for locks held by the first transaction. Locking algorithms can be divided in to two classes: static locking and dynamic locking. In static locking, all lock requests of a transaction are allotted at the same time, and execution does not start until all requests are granted. In dynamic locking, a lock request is issued whenever a data item is needed. In a distributed database system, where transmission delay may be substantial, static locking schemes are preferable to dynamic ones [SHY90]. This is so because static locking schemes allow concurrent transmission of lock requests, and improve the response time of transactions.

The deadlock detection schemes, proposed with the deadlock detection algorithms [CHA82, CHA83, GLI80, HAA83, MEN79, OBE82, SIN85] call for abortion(restart) of some transactions in the deadlock cycle. Deadlock detection is difficult in a distributed database system, because no site has a complete and upto date information about the system. Some algorithms detect deadlocks by first constructing and then finding cycles in a transaction-wait-for-graph (a directed graph whose nodes represent transactions and arcs represent the wait-for relationships). Solutions based on this method, are quite expensive because a large amount of information needs to be propagated from site to site [OBE82]. In static locking, in case of deadlock, the restarted transaction must release all its locks and then send out the access requests again. In a high volume transaction processing environment this degrades database performance dramatically. The performance study [CHO90] indicates that the major component of the cost of running the algorithms[HAA83, SIN85] occurs when there is no deadlock, i.e., the cost of running the algorithm, dominates the cost of running the algorithm when deadlock does exist. In the algorithm proposed in [SHY90], the deadlocks are resolved by reordering the lock requests of the data items, such that no transactions need to be aborted. In this approach, the algorithm must be run regularly, to detect deadlocks.

In this paper, we have considered a deadlock free approach based on formation of precedences by site data managers, for accessing data items. A lock request of transaction T_i, is sent to different sites. At each site, it forms the local access graph(LAG). A LAG of T_i at site S_j contains the precedences of all transactions(T_j) such that both T_i and T_j have a conflict on some data items resident at S_j. The formation of LAG at different sites is based on the transaction identification numbers, which are assigned uniquely to each transaction at the site of their origin. After forming the LAG, the odd precedences are detected and

readjusted, if necessary by consulting the respective transactions' home site. In this algorithm, extra communication is needed, if and only if, odd precedences exist in a respective LAG. The approach causes no deadlocks, and hence does not lead to an abort or a rejection of a transaction.

In the next section, we define a system model, along with some definitions. In section 3, the synchronization technique is described in detail. Section 4, gives the algorithm for formation of a local access graph at various sites. In section 5, proof of correctness is considered and the last section considers the summary and conclusions.

2.0 SYSTEM MODEL AND RELATED TERMS

We assume that, a distributed database management system consists of a collection of sites connected by a computer network. A database is a collection of data items. Each site has a system wide unique identifier. Further we assume that each site supports a transaction manager(TM) and a data manager(DM). The TMs supervise execution of the transactions while the DMs manage individual databases. The network is assumed to detect failures, whenever these occur. When a site fails, it simply stops running and other sites detect this fact. The communication medium is assumed to provide the facility of message transfer between sites. When a site has to send a message to some other site, it hands over the message to the communication medium, which delivers it to the destination site in finite time. We assume that, for any pair of sites S_i and S_j, the communication medium always delivers the messages to S_j in the same order in which these were handed to the medium.

A transaction is modelled as a sequence of read and write operations. A transaction consists of four steps: reads, local computations, writes and transaction commit. The lock request messages are sent to the corresponding sites. Each site assumed to support both local as well as global transactions. The notion of correctness of transaction execution in this context, is that of serializability[ESW76]. We assume two phase static locking is used. After a lock is granted, a lock grant message(with the data) is sent to the transaction generation site. Local computation starts after all lock grants are received. After the local computation is performed, the write phase is initiated. Updated data items are sent to the data sites and are stored in some temporary memory. In the commit phase, updated values are written into the database. Locks are released at the end of the commit phase. For the study, all lock requests are considered to be for exclusive access to data items. The items to be locked by the transaction for the purpose of read/write steps are termed as, locking variables(LVs). The transactions T_i, T_j are said to have conflict if, $LV(T_i) \cap LV(T_j) \neq \phi$. In this paper, if we say $T_i \cap T_j \neq \phi$, to imply that conflict exists between T_i, T_j.

2.1 Transaction Identification Number (TIN)

Every site S_i has a logical clock C_i, which takes a monotonically nondecreasing integer value [LAM78]. A TIN is assigned to the transaction T_i, on its arrival, by a site S_i. It is a triple element value, as (S,I,C). S is the site identifier. The I field is the unique transaction identifier, which is a value of the local clock (C_i) at the instant of T_i's arrival to the home site S_i. The C field is used to synchronize different clocks. It does not contribute to the identity of the transaction.

For any transaction message, T(S,I,C), about to be sent

C: = C_i of the message sender site.

C_j of the site S_j, on receipt of a transaction T(message)
$C_j := \max(C+1, C_j)$.

C_j of the site S_j, before dispatch of a transaction T(message)
$C_j := C_j + 1$.

Let, TIN1 = (S_1,I_1,C_1) and TIN2 = (S_2,I_2,C_2) then, any pair of TINs can be compared using the following criteria.

Equal to(=) : $TIN_1 = TIN_2$, if and only if, $S_1 = S_2$ and $I_1 = I_2$;

Greater than(>) : $TIN_1 > TIN_2$, if and only if, $I_1 > I_2$, or $I_1 = I_2$ and $S_1 > S_2$; and

Less than(<) : $TIN_1 < TIN_2$, if and only if, $I_1 < I_2$, or $I_1 = I_2$ and $S_1 < S_2$.

2.2 Partial Order(<<)

Consider T, as a set of transactions. A partial order L = (T, <<) consist of set T, called the domain of the partial order, and an irreflexive transitive binary relation << on T. If $T_i << T_j$, we say that T_i precedes T_j in execution.

3.0 SYNCHRONIZATION USING LOCAL ACCESS GRAPHS (LAGs)

Let, $T = \{T_1, T_2, ..., T_n\}$, be a set of active transactions in the distributed system. A local access graph of T_i at site S_i is a partial graph G(V,E), where $V \epsilon T$, and $E = \{ <T_i, T_j> / T_i \cap T_j \neq \phi$ and $T_j << T_i \}$. In this $T_i \cap T_j \neq \phi$ denotes both T_i, T_j has the conflict on some data item resident at S_i.

When a lock request of T_i is sent to the site (S_i), a LAG is constructed at S_i. The notion of LAGs is described with an example given below.

Example 1: Consider the transactions T_1, T_2, T_3, T_4 and T_5 as shown below. Let X,Y,Z be data items, and w(X) indicates the write operation on data item 'X'. Let, transaction requests be as:

$T_1 = w_1(X)\ w_1(Y)$
$T_2 = w_2(X)\ w_2(Y)\ w_2(Z)$ $T_3 = w_3(Z)$
$T_4 = w_4(X)\ w_4(Z)$ $T_5 = w_5(Y)\ w_5(Z)$

Consider the case, Where X,Y and Z are located at one site. The execution of above transactions can follow any sequence. The notion of correctness of transaction execution is that of serializability. So, for above transactions many LAGs are possible, if we take different serial executions in which, transactions may execute operations. If we take arrival pattern of transactions in the order T_1, T_2, T_3, T_4 and T_5, then based on the arrival pattern, the corresponding LAGs of above transactions are shown in figure 1.

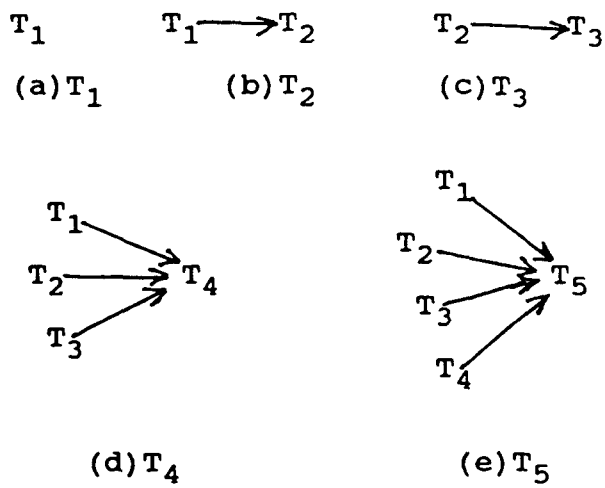
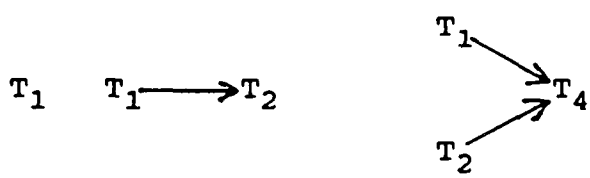
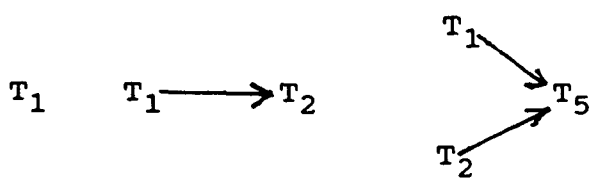


Figure 1

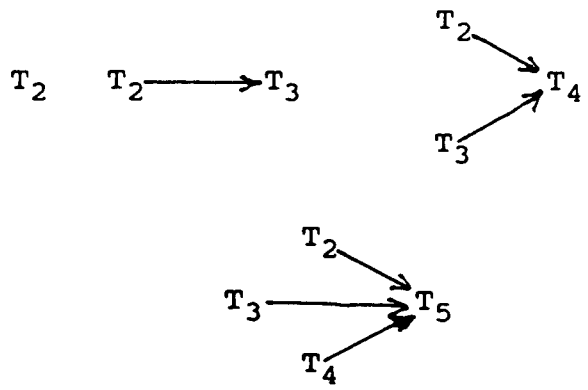
Now consider the case of a distributed system where data items X,Y and Z are stored at S1, S2 and S3 respectively. The LR of a transaction forms LAGs at sites where its conflicting data items reside. Now the LAGs of each transaction at site is shown in the figure 2. The home sites of T_1 is S_1; T_2, T_5 is S_2; and T_3, T_4 is S_3.



(a) LAGs at S_1
(locks are granted to T_1)



(b) LAGs at S_2
(locks are granted to T_1)



(c) LAGs at S_3
(locks are granted to T_2)

Figure 2.

As seen above, in the centralized version (figure 1.) the LAG of T_i contains precedences of all active transactions which have any conflict with T_i. But in distributed version (figure 2.), it is not possible to have all precedences included in the LAGs formed by an individual site. Inspite of this, the use of LAGs is similar to locking. In the case of static locking, the lock requests wait in queues. In the proposed approach, LAGs are constructed instead of lock queues.

In figure 2, edge$<T_i,T_j>$ denotes that T_j requires some data item which is also used by T_i. After execution of T_i the conflicting data item is released, for access by T_j. At the beginning, T_1 accesses the database of S_1. After this, T_2 accesses the database at S_2, then T_3, T_4 at S_3 and T_5 at S_2. Thus the equivalent serial order, obtained is thus $<T_1,T_2,T_3,T_4,T_5>$.

4.0 AN ALGORITHM TO CONSTRUCT LAG

For a transaction T_i, at site S_i, the locking variables of T_i are identified. A lock request is prepared and is sent to corresponding sites. At the concerned sites, the LAGs are formed.

4.1 Definitions

Home site:
The originating site of transaction T_i is called as the home site of T_i.

Odd edge, Even edge:
An edge$<T_i,T_j>$, such that, $T_i > T_j$, is called as odd edge. Otherwise, edge $<T_j, T_i>$ is called an even edge.

Transaction identification number (TIN):
This is a unique number (S,I,C), assigned to the transaction T_i on its arrival, by the home site. In this paper T_i, T_j,.. represents the TINs of corresponding transactions.

Locking variables(LVs):
For a transaction T_i, the items read, or to be written by T_i constitute the locking variables.

Lock request(LR):
It consist of TIN, LVs. It is prepared by home site, on arrival of each transaction.

Data table(DT):
This table is maintained at the home site of each transaction. The DT of T_i contains the lock grants (with values). Whenever S_i receives the lock grant of T_i, from other site, it stores in the T_i.DT.

Transaction status(TS):
TS supports two values, 0 or 1. After getting all required lock grants, transaction T_i changes the T_i.TS to 1, then starts execution. Otherwise, T_i.TS = 0. The value of T_i.TS is maintained at the home site(the requesting TM).

Access status(AS):
It has values, 0 or 1. After access to the required data items at some site S_j, by T_i, the T_i.AS becomes 1. Otherwise it is 0. This parameter is maintained by the site S_j.

Active transaction list(ATL):
The ATL is maintained by each site. ATL is divided in to two tables ATL.T and ATL.G.

ATL.T of S_i = $\{T_i(TVs,TS,AS) \mid T_i$ has requested data items resident at $S_i\}$. Where T_i may be a local or global transaction.

ATL.G of S_i = $\{LAG(T_i) \mid T_i$ has requested data items resident at $S_i\}$.

4.2 Algorithm

The following notations are used to describe the algorithm.

T_i, T_j,...represents the TINs of corresponding transactions.
T_i.home-->home site of T_i.
T_i.TS -->TS of T_i.
T_i.AS -->AS of T_i.
S_i.LAG.T_j-->LAG of T_j at S_i.
T_i.DT-->DT of T_i.

In this algorithm, whenever a transaction arrives at S_i, it is assigned a TIN value. Let Ti be the TIN. Its LR is prepared and is sent to all sites where the data items reside. The LAG is prepared at concerned sites. At any site S_j, if S_j.LAG.T_i contains odd edge$<T_j,T_i>$, then it is confirmed by checking the locally existing S_j.LAG.T_j, or by consulting the T_j.home. That is, either the lock request has not been granted and the odd edge can be reversed locally. Or, if the lock request has been granted, the transaction may be in execution or it may not be in execution. That is, at the T_j.home, if T_j is under execution, then the odd edge$<T_j, T_i>$ is not deleted from the S_j.LAG.T_i. Otherwise, the even edge$<T_i,T_j>$ is inserted into the S_j.LAG.T_j, and odd edge$<T_j,T_i>$ is deleted from S_j.LAG.T_i. In this way all odd edges are confirmed, resulting in a deadlock free environment. The algorithm is described below.

I FORMATION OF LAGs.

(1) Processing at home site

On arrival of a transaction at site S_h, the site S_h assigns it a T_i(TIN). It also identifies the LVs. The S_h prepares the LR for T_i as: (T_i,LVs). The LR is sent to different sites.

(2) Processing at any site

A LR of T_i is received by S_i. The following steps are carried out at the site.

Initialize the S_i.LAG.T_i(V,E) as: V={T_i} and E=ϕ. Insert the edge $<T_i,T_i>$ into the S_i.LAG.T_i for all conflicting transactions(T_i) from the ATL.T of S_i. If T_i.home$\neq S_i$ then insert the T_i(LV,AS) into the ATL.T with T_i.AS=0. If T_i.home=S_i, then insert T_i(LV,TS,AS) into the ATL.T with Ti.TS=0 and Ti.AS=0. Store Si.LAG.Ti into ATL.G. Go to (3).

II CONFIRMATION OF ODD EDGES AND LOCK GRANT

(3) For each odd edge$<T_k,T_i>$ from S_i.LAG.T_i go to (3.1). After completing all odd edges go to (3.3).

(3.1) If T_k.AS=0, then insert the edge $<T_i,T_k>$ into the S_i.LAG.T_k. Delete the edge$<T_k,T_i>$ from S_i.LAG.T_i. If T_k.AS=1 then go to (3.2).

(3.2) If T_k.AS=1 then send the message 'T_k.STATE' to T_k.home. In return to this message, S_i receives the message 'T_k.EXECUTION', then do nothing. If S_i receives the message 'T_k. NOEXECUTION' then insert the edge $<T_i,T_k>$ into the S_i.LAG.T_k. Delete the edge$<T_k,T_i>$ from S_i.LAG.T_i. Go to (3).

(3.3) If T_i receives the commits of all transactions (T_j), such that, $<T_j,T_i>$ belongs to S_i.LAG.T_i, then change the T_i.AS to 1; and send the lock grants(with the values) to T_i.home.

III COMMIT PROCESSING OF TRANSACTION AT THE HOME SITE

(4) T_i waits until it gets all the requested locks. All lock grants are stored in the T_i.DT. After getting the all lock grants, the site assigns status T_i.TS as 1 in the ATL.T and starts execution of T_i. After completion of execution, the updated values are committed.

(5) Whenever S_h receives the message 'T_k.STATE' from S_i, if T_k.TS=1, then the message 'T_k.EXECUTION' is returned to S_i. If T_k.TS=0, then the lock grants entry of T_k from S_i are deleted from T_k.DT, and the message 'T_k.NOEXECUTION' is sent to S_i. Thus, the requesting site can revoke locks already granted, to avoid a deadlock.

5.0 PROOF OF CORRECTNESS

Theorem1: According to above algorithm, the execution of transactions follows some partial order(< <).

Proof: We prove the following

(i) every pair of conflicting transactions forms a precedence.
(ii) deadlocks do not occur.

(i) if T_1, T_2 are conflicting type, then both will make a request to the site where conflicting data items are available. Therefore, a precedence is formed between T_1 and T_2.

(ii) Consider the deadlock situation as:

$$C=T_1 \xrightarrow{w} T_2 \xrightarrow{w} \ldots \xrightarrow{w} T_n \xrightarrow{w} T_1;$$

where, n>1. In this, $T_i \xrightarrow{w} T_j$; indicates T_i is waiting for data item 'X', which is locked by transaction T_j. Among the given edges, for any odd edge $<T_j,T_i>$,

(a) If T_j is under execution, then the cycle breaks.
(b) If T_j is not under execution, then the edge $<T_j,T_i>$ is deleted(odd edge).

From (a) and (b), it is proved that C is acyclic.

From (i), every pair of conflicting transactions form precedence and from (ii), cycles do not exist. Hence the execution of transactions is as per the partial order(< <).

6.0 SUMMARY AND CONCLUSIONS

In the existing locking based approaches, if transactions from different sites, are in serializability conflict, then some of the submitted transactions are rejected. In these systems deadlock removal requires extra messages, and processing time. Also, the transactions are resubmitted for execution, and by this, incur additional processing delays and overheads. In the proposed technique, a local access graph is constructed at every site, which results in prevention of deadlocks. Given an ideal situation, if each data access request is made as per the TIN ordering, the chances of a deadlock are extremely low. In the case of confirmation of odd edge, in many cases, the confirmation is done locally.

In the proposed algorithm, transaction rejection, or abort does not occur. The proposed technique is ideally suited for a distributed environment, where a dominant factor of cost of processing is, number of messages. The proposal can be implemented with slight changes to existing locking based mechanisms. No separate deadlock handling mechanism, needs to be implemented in such systems.

REFERENCES

[CHA82] Chandy,K.M and Misra, J. A Distributed algorithm For Detecting Resource Deadlocks in Distributed Systems. Proc. of ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, Ottawa, Canada, Aug. 1982.

[CHA83] Chandy, K.M., Misra, J. and Hass, L.M. Distributed Deadlock Detection. ACM Trans. on Computer Systems. Vol.1, pp. 144-156, May 1983.

[CHO90] Choudhary, A.N. Cost of Deadlock Detection: A Performance Study. IEEE Conference on Data Engineering, 1990.

[ELM86] Elmagarmid, V.D. A Survey of Distributed Deadlock Detection Algorithms. SIGMOD RECORD, 15(3), pp.37-45, Sept. 1986.

[ESW76] Eswaran, K.R., Gray, J.N., Lorie, R.A and Traiger, T.L. The Notions of Consistency and Predicate Locks in a Database System. Communications of ACM, Novem. 1976.

[GLI80] Gligor, V and Shattuck, S.H. On Deadlock Detection in Distributed Database Systems. IEEE Transactions on Software Engineering, VOL. SE-6, Sept. 1980.

[HAA83] Haas, L.M and Mohan, C. A Distributed Deadlock Detection Algorithm For Resource Based Systems. IBM Research Report, RJ3765, Jan 1983.

[KNA87] Knapp, E. Deadlock Detection in a Distributed Databases. ACM Computing Surveys, 19(4), pp.303-328, Decem. 1987.

[LAM78] Lamport, L. Time Clocks and Ordering of Events in a Distributed System. Communications of ACM, 21(7), pp.558-569, July 1978.

[MEN79] Menasce, D.A. and Muntz,R.R. Locking and Deadlock Detection in Distributed Databases. IEEE Transactions on Software Engineering, Vol. SE-5, May 1979.

[OBE82] Obermark, R. Distributed Deadlock Detection Algorithm. ACM Transactions on Database Systems, Vol. 7, pp.187-208, June 1982.

[SHY90] Shyu, S.C., Li, V.O.K and Weng, C.P. An abortion free Distributed Deadlock Detection/ Resolution Algorithm. Proc.IEEE 10th International Conference on Distributed Computing Systems, June 1990.

[SIN85] Sinha, M.K and Natarajan, N. A Priority Based Distributed Deadlock Detection Algorithm. IEEE Transactions on Software Engineering, Vol. SE-11,pp. 67-80, Jan 1985.