90-RC

ISMM

INTELLIGENT DISTRIBUTED PROCESSING

Ft. Lauderdale, Florida
December 13-15, 1989

EDITOR: R. Ammar

A Publication of
The International Society for
Mini and Microcomputers - ISMM

ISBN 0-88986-138-2

ACTA PRESS

ANAHEIM * CALGARY * ZURICH

ISMM

Proceedings of the ISMM International Conference, Intelligent Distributed Processing, held in Fort Lauderdale, Florida, U.S.A. December 13-15, 1989.

Editor: R. Ammar

Copyright © ISMM

ACTA PRESS
P.O. Box 2481
Anaheim, CA
U.S.A. 92814

ACTA PRESS
P.O. Box 3243, Stn. "B"
Calgary, Alberta
Canada T2M 4L8

ACTA PRESS
P.O. Box 354
CH-8053 Zurich
Switzerland

ACTA PRESS CODE: 152

A RECOVERABLE AND CONCURRENT LOCKING ALGORITHM FOR A PARALLEL SEMANTIC BINARY DATABASE MACHINE WITH INVERTED REPLICATION [1]

Ragae Ghaly *Naphtali Rishe*

School of Computer Science
Florida International University
University Park
Miami, Florida 33199

Abstract

A recoverable and concurrent locking algorithm is proposed for the Linear-throughput Semantic Database Machine (LSDM), a multi-disk, multi-processor database machine that offers massive parallelism. The database environment is based on the Semantic Binary Model (SBM), a fact-oriented representation of an information system. The whole database is represented by a set of facts and their inverted replicas, which are distributed evenly with balanced segments among the available processors. Our algorithm employs a locking strategy based on time-stamp ordering of transactions. The requested locks depend on the granularity of data which is selected to be either a fact or an object (a contiguous range of facts associated with one entity). The algorithm is deadlock free since transactions are globally ordered among all sites. Fairness and freedom from starvation are ensured, since transactions are partially ordered over the waiting queues of a particular site. The recovery algorithm monitors and controls the execution of transactions so that the fact base includes only the results of the committed transactions.

Keywords: transaction, parallel processing, scheduling, database

1 INTRODUCTION

Systems that solve the concurrency control and recovery problems allow their users to assume that each of their programs (transactions) executes in an atomic (serializable) and reliable (recoverable) manner. That is, each transaction accesses shared data without interfering with other transactions; and if a transaction terminates normally, then all its effects are made permanent, otherwise it has no effect at all.

The Semantic Binary Model (SBM) [6,7,9,10], is a fact-oriented representation of an information system. It supports aggregation, classification and generalization abstraction mechanisms [5]. Objects of the real world are represented as abstract objects or printable objects. SBM represents information of the Universe of Discourse (UoD) as a collection of elementary facts of two types:

- *Unary facts:* that categorize the objects of the UoD into categories (entity sets). Categories can intersect. Properties are inherited by subcategories. The fact aC means: the object a belongs to the category C.

- *Binary facts:* relate object-pairs in relationships. A relation can be 1:1, 1:M, M:1 or M:M. The fact xRy means that there is a relation R between objects x and y.

A storage structure [8] for SBM provides for a highly efficient performance of simple queries. An elementary query normally requires only one disk block access. Every abstract object in the database is represented by a unique object identifier (OID). Various categories and relations of the schema are also treated as abstract objects. Elementary retrieval operations are listed in [8]. Among them:
a?, ?C, aR?, ?Ra, a?+a??+??a, ?Rv, ?R[v1, v2].
(The elementary range query: *?R[v1, v2]* finds the objects x related by R to a value v where $v1 \leq v \leq v2$).
In order to expedite the query/transaction processing, the entire database is stored in a single file that contains all facts and their inverses sorted in a lexicographical order. The inverse fact of aC is $\check{C}a$; the inverse of xRy, where x and y are abstract objects, is $y\check{R}x$; the stored inverse of xRv, where v is a value, is $\check{R}vx$.
The file is maintained as a B-tree that allows both sequential and random access. All facts related to an object are clustered together in the sorted file.

The proposed locking algorithm is applied to the Linear-throughput Semantic Database Machine (LSDM) [8]. LSDM is a multi-disk, multi-processor database machine which offers massive parallelism. The architecture is designed to allow large databases with high query throughput and to adapt linearly to any further increases in the number of the processors and/or disks. It also enables load balancing among the processors. All processors are assumed to be identical and connected by a hypercube-like network via high speed communication channels.

First, we shall discuss the fact distribution process in the LSDM. Then, in Section 3, an object-oriented design for the transaction model is discussed. In Section 4 we discuss the proposed concurrent and recoverable algorithm.

2 FACT DISTRIBUTION PROCESS

The whole database is represented by a file of facts and their inverses, sorted in lexicographical order and sliced into evenly balanced segments of facts[12]. The number of segments is equal to the number of processors. The database file is distributed among n sites. Each site is charged with managing all the original facts that belong to a specific set of objects. Those sites are called the primary sites since they holds the primary copies of facts. Inverted replication of all facts is included in the distribution process in the sense that only one copy (the primary copy) is at one site and its inverted one is (normally) at another site. A copy of the partition table called Object Map Table (OMT) is kept at each site in order to associate each object/fact with its hosting site. The inverse replication of facts assists in fast and efficient manipulation of elementary queries. Fact distribution process can be viewed either by:

- using only one fact segment type.
- using two fact segment types:
  - one for the regular facts, that is the *primary* and *inverted* facts beginning with an abstract object: aC, aRb, $b\check{R}a$, aRv
  - another for the other inverted facts: $\check{C}a$, $\check{R}va$.

    where
    a, b: abstract objects
    v: concrete object (value)
    C: category (unary relation)
    R: a binary relation (between two objects)
    $\check{C}$: is the inverse of C
    $\check{R}$: is the inverse of R.

It is assumed that the distribution process is optimized in the sense that each site will be concerned with all the information regarding some regular objects. On the other hand the irregular facts *($\check{C}a$, $\check{R}va$)* will be sliced and divided among different sites.

In case of an expected overflow of facts at one of the site's disk storage, a redistribution process must be established to rebalance the load of facts

[1] This work has been supported in part by a grant from Florida High Technology and Industry Council.

among all the sites. This process is quite expensive and should be handled with complete transparency to the system's users. Database management functions are replicated in each site, and each site can be acting as an originator of a certain transaction and/or contracting transactions to the appropriate sites. Each site can be acting as a scheduler to execute those subtransactions from different sites that are associated with the that fact base.

Data granularity is structured for a fact or an object. The coarse granule in our system is considered to be an object and the finer granule is a fact. This convention is important in the definitions of further steps in the algorithm, specifically in the locking strategy. However, granularity is a performance issue, it improves the concurrency by allowing a transaction to lock only those facts it accesses, but the fine granularity also involves higher locking overhead, since more locks are requested [1].

3 THE TRANSACTION MODEL

A transaction is a set of interrelated queries/updates of objects/facts that accesses a shared database and is executed atomically [10,11]. Syntactically, a transaction T consists of a set of queries and deletions and insertions of objects/facts, bracketed by a Start command at one end (when T begins its execution) and either Commit or Abort command at the other end. An example of a transaction is shown in Fig (2). Our transaction model Fig (1) consists of a distributed Transaction Manager (TM), Scheduler or Lock Manager (LM), and a Fact Manager (FM), that are located at each site [1].

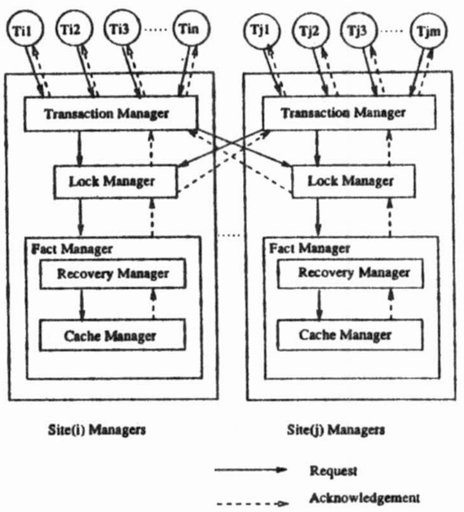
3.1 General assumptions

- All sites are identical in their abstraction functions.
- All modular abstractions either of one site or different sites interact with each other through the technique of ***handshaking***, in order to ensure ordering of operations by requests and acknowledgements using a message passing technique.
- Each transaction has a ***unique identifier*** that is generated by the system and is transparent to the user. The identifiers establish a global order of transactions. This assumption can be enforced by using the local clock number generated by the system as a prefix that is concatenated with the site-id to form the transaction's global identifier [3].
- A query will be treated at the same priority level as other update transactions.
- In order to avoid cascading aborts, only committed transactions can affect other transactions .
- Site failures are detected by a ***timeout***.
- Each transaction preserves its own consistency. Consistency and integrity is a duty of the TM.

3.2 Transaction Manager Abstraction

TM acts as an interface between the user transactions and the database system. TM assumes that the user calls on any site without a prior knowledge of the availability of facts at that site. TM must take charge of that transaction and coordinate its execution among the sites concerned until all the results are obtained and are passed to the user. TM is aware of each site's availability and the range of facts hosted by that site (from the local Object Map Table).

- TM manipulates each transaction as an atomic, self-contained unit whose execution is transparent to the other user's transactions.
- Transactions are assumed to be compiled by the TM functions before its execution (before the TM issues a Start command). The compiler can map requests efficiently by generating the appropriate lock type for each request. This simply can be established by allowing the compiler to consult the Object Map Table and to apply the majority-rule concept in its decision. The lock type is decided as far as granularity is concerned.
- The TM communicates only with LM schedulers at either the same site or at different sites. No communications is assumed between



Fig(1): Communications between Modular Abstractions of Different Sites

TM$_i$ and TM$_j$ at two distinct sites [4].

- The TM uses atomic commit protocol (ACP) that can consistently terminate a transaction that access data at more than one site.
- The TM follows the Read-any-Write-all approach [1], in the sense that the read-requests require to lock either the primary or the inverted fact, but the write-requests require to communicate with both the primary and the inverted facts.
- The TM enforces the Strict Two-Phase Locking (2PL) by being responsible for requesting all required locks from LMs during the transaction's lifetime at the *growing phase*, until it sends its Commit/Abort command announcing to each invoked LM that the growing phase that ended and no more locks may be requested later. The TM is not responsible for requesting to release locks, since this is a strategic function of the LM.

Operations:

***Start, Commit, Abort*:** control commands for transaction management.

***Read, Delete, Insert*:** for object/fact management.

***Compile, Optimize, Generate, Verify* :** operations assisting in making decisions regarding commit or abort at transaction, generation of proper lock types, checking validity and integrity constraints.

```
Start
    q(x C1)
    q(x R1 y)
    Q(z)
    Q(R1)
    d(y R2 15)
    D(u)
    i(w R1 y)
    I(v)
Commit/Abort
```

q:	read a fact
Q:	read all facts of an object
d:	delete a fact
D:	delete all facts of an object
i:	insert a new fact
I:	insert group of facts of an object
C1 :	abstract category object
R1,R2 :	abstract relation objects
u,v,w,x,y,z:	abstract objects

Fig(2) : A Sample of an Optimized Transaction

3.3 Lock Manager (Scheduler) Abstraction

The Lock Manager of each site manages the scheduling of all lock request operations and guarantees the transactions' execution in a serializable and recoverable manner. The processing of data within a subtransaction of one site is transparent to all the other (sub)transactions of the other sites.

- LM$_i$ is only sensitive to conflicts between operations on the same copy of that fact/object hosted in its site. All the LMs are working independently.
- The LM of a site communicates with the only FM of that site in

order to execute the transaction's operations.

- The LM takes part in enforcing the Strict 2PL protocol only at the *shrinking phase* by being responsible for releasing the locks. This policy can let the LM work independently and, thus, avoid any communication with other sites' LMs. Since each LM_i obtains the complete information, it can decide when to process an operation without any communication with other sites.
  - LM_i can not release locks of T_j until it knows that TM_j will not submit any more lock operations to any other LM. This is to enforce the 2PL rules. Fig (4) shows the proposed behavior of the locking strategy.
  - In order to avoid starvation and lockout situations, LM keeps a waiting queue ordered by the global T-id. This insures fairness in selecting the next transaction to run.
  - The LM performs a part of the log operations while committing a subtransaction at that site, and it also guarantees that the recoverability conditions hold for all transactions' read operations at that site.
  - LM controls the concurrent execution of the interleaving transactions by restricting the order in which the FM executes Read, Delete, Insert, Commit and Abort of different transactions.

Operations:

- *Lock (T-id, object/fact, mode)*
- *Unlock (T-id, object/fact)*

3.4 Fact Manager Abstraction

Fact manager keeps track of the physical changes in the database: deletion and insertion of facts. It also manages the preparation at the physical locks on objects/facts before acknowledging to the LM. This abstraction consists mainly of two abstractions, the Recovery Manager (RM) and the Cache Manager (CM) [1]:

3.4.1 Recovery Manager Abstraction

A recovery algorithm monitors and controls the execution of programs so that the fact base includes only the results of committed transactions. If a failure occurs while a transaction is executing, and the transaction is unable to finish executing, then the recovery algorithm must wipe out the effects of the partially completed transaction. Thus, it ensures that the fact base dose not reflect the result of a partially committed or aborted transaction. It also ensures that the results of the successful execution of a transaction are never lost.

Thus, RM maintains a log of operations and guarantees that all the transactions' write operations are logged on a stable storage before acknowledging the LM (Redo Rule)[1]. The RM ensures that a transaction may read only those values that are written by committed transactions or by itself, in order to avoid cascading aborts. The RM guarantees also that no fact may be read or overwritten until the transaction that previously wrote into it terminates, either by Abort or Commit.

Operations:
Start, Commit, Abort, Complete, Read, Write

3.4.2 Cache Manager Abstraction

The Cache Manager assists in moving facts/objects between volatile and stable storage for requests from the LM of the same site [1].

Operations:

- *Fetch(z):* retrieves a certain fact/object from the stable storage into the volatile storage.
- *Flush(z):* stores a certain fact/object into stable storage from the volatile storage.

4 THE LOCKING ALGORITHM

We have two levels of locking granularity: a fact or an object. Multigranularity locking allows each transaction to use granule sizes most appropriate to its mode of operation.

- Long transactions, those that access ranges of facts may request object locking (coarse granule).
- Short transactions, may request fact locking (fine granule).

A concurrency control algorithm ensures that transactions are executed atomically. It does this by controlling the interleaving of concurrent executions of transactions, to create an illusion that transactions execute serially, one after the next, with no interleaving at all. The concept of serializable transaction is the basis on which any algorithm ensures correctness of interleaved operations.

A good concurrent algorithm should be correct in the sense that it is deadlock free and starvation free. In the proposal of that algorithm we assumed this correctness due to the fact that all transactions are globally ordered and LMs refuse those transactions that arrive late to the site and are requesting a conflicting lock that has already been granted to a younger transaction. Thus the algorithm ensures the serializability.

4.1 Transaction Manager

When a transaction T arrives, certain tasks must be processed sequentially in order to prepare for the transaction execution. The transaction will be assigned a new global T-id according to the local clock time. The effect of the clock on the system's performance will be discussed later. In our locking algorithm, once the TM requests a lock from any LM, it is waiting to receive an acknowledgement before sending another request to that particular LM.

The following requests are issued from the TM to the concerned LM:

- TM prepares the execution of the transaction T by sending a Start command to all the LM_j where the j's are decided upon completion of the lock generation phase of the compiler and include each site involved in executing T. The TM waits for acknowledgements from each site involved in the transaction. For the sake of recovery, the TM records <Start T-id> on its own log.
- Depending on the granularity level, each Read request will be sent to the appropriate primary LM. The request is implicitly accompanied with a shared lock. Thus, the following implicitly *shared* locks are defined and sent to LM_j
  - *q(f):* request a read lock for a fact *f*.
  - *Q(X):* request a read lock for all the facts belonging to an object *X*.

  In case of q(f), if a failure of primary site is detected (by timeout), it can be reasonable to request this single fact from the inverted site. This might be an expensive process for a query about an object since a request must be sent to all the inverted sites involved.
- Deletion of a certain fact or an object also depends on the granularity level. The following requests for *exclusive* locks are sent to both LMs at the primary and inverted sites:
  - *d(f)* : request a lock for the deletion of fact *f*.
  - *D(X)* : request a lock for the deletion of object *X*.
- Insertion of a new fact/object. It is necessary to lock the new fact, before its insertion, in both the primary and inverted sites, so as to prevent any conflict with a read/delete. The explicit lock request from TM to LM indicates that necessity. The following *exclusive* lock requests from both LMs at the primary and inverted sites are sent:
  - *i(f)* : request a lock to insert a new fact *f*.
  - *I(X)* : request a lock to insert a new object *X*.

In the above two cases (delete/insert) a transaction T may decide to abort and rerun due to a failure detection of any site involved in executing T.

While the transaction is active, the TM collects the necessary information to check database integrity. After all the local requests of (q, Q, d, D, i, I) are performed and all the necessary locks are obtained by the LM of each site, the transaction performs its integrity constraint tests. Depending on these results, the transaction sends either a Commit or an Abort command. At this stage the transaction is considered *terminated* or partially committed.

When TM does not receive any acknowledgement from an LM due to a timeout for any request other than Start, it will assume that the site is dead and will have to perform the following routine:

- Update the available site list and reconfigure its connection paths to the available object/facts.
- If the failed site holds only a shared lock, it can be found from its inverted site. Depending on the inverted facts that can be collected for a shared object from different sites, TM can decide to continue on that strategy or to abort.
- If the failed site holds an exclusive (delete/insert) lock on a fact/object then it is necessary to dispatch an Abort to all other sites that are involved in the transaction to allow other waiting T's to continue execution.

Similarly, TM can receive a late reply from an already *assumed* failed site: either it was due to a traffic jam on the link that caused the delay or due to a link failure that could not be recognized by either the LM of the locking site or the TM of the requesting site. Another possible message can be received from a recovered site that has just come to life and is only sending messages to those TM_j that were on its incremental log. TM must take care of these situations, by updating its available site list.

4.2 Lock Manager

The lock manager is a distributed abstraction that keeps track of the locks that are issued to transactions. It does this by using a local Lock Management List (LML), that consists of a two level data structure. At the top level there is the Object Lock Header (OLH), which keeps track of the explicit locks at objects or implicit locks at facts at the bottom level. The bottom level consists of the Fact Lock Table (FLT) that keeps track of the explicit locks on facts that belong to one of the top level objects. An entry of OLH consists of an OID, object lock mode (shared/exclusive), the fact-range of that object, the number of facts locked explicitly in the FLT that belong to that object, and a pointer to the Wait-For-Queue(X) that holds all T-ids requesting that object that are in conflict with the existing lock. Also, it tells which T-id is utilizing the lock and a list of all T-ids that are compatible and are utilizing the lock at the same object.

Each entry of the FLT consists of the fact, the mode (shared/exclusive) and a pointer to a Waiting-For-Queue WFQ(f) that keeps track of each T-id that is waiting for a lock on that fact. It tells also which T-id is compatible and is utilizing the lock at the same fact. Locking management must also be very fast since it consumes a significant fraction of the processor's time. As a common practice, lock entries are hashes to either table in order to enhance the looking up mechanism. Also, the LM links all of the read lock entries in the LML_i, and all of the write lock entries of each transaction together to speed up the releasing action of locks.

Another data structure that keeps track of each transaction status within the site is the Transaction Header Table (THT). It tells how to trace a transaction's locks among the waiting queues, in order to speed up the lock release mechanism. Also, the THT should keep track of the time a transaction has spent inside the system, in order to prevent starvation of other transactions waiting for lock releases. An example of the Lock Manager's data structures snapshot is shown in Fig (4).

A complete analysis of the LM behavior is as follow:

- Every lock request is fairly granted according to the availability of the fact/object as decided by the FM_i.
- Once a lock is granted, it is kept in the LML_i, and an acknowledgement is issued to the appropriate TM_j. A timeout limit (predefined, but fairly different than that of TM_j) is set for that transaction in THT_i. This prevents the starvation of the waiting transactions for a lock held by a failed site.
- Two lock requests are in conflict if one holds an exclusive lock according to the compatibility matrix as shown in Fig (3).
- A transaction T_j that requests a lock that conflicts with a granted lock waits in the WFQ_i of that object/fact. Each WFQ is ordered by T-id.
- When lock type is requested on a specific fact, the appropriate intention lock is set first on its object at the OLH, then the fact is locked explicitly at the FLT. In our implementation, all facts are assumed to have an explicit lock, while objects can have either an intention

	q	Q	d	D	i	I
q	T	T	F	F	F	F
Q	T	T	F	F	F	F
d	F	F	T	T	F	F
D	F	F	T	T	F	F
i	F	F	F	F	T	F
I	F	F	F	F	F	F

Fig(3) : The Compatibility Matrix of Requesting Locks

lock (due to locking same facts) or an explicit lock.

- When TM_j requests a Commit/Abort, the same message is sent to the FM_i to execute it and the LM_i start releasing all read locks to ensure serializability (when the transaction terminates), but write locks are only released when the FM_i acknowledges a Commit/Abort. This enforces the 2PL in a distributed environment, since the TM_j of a particular transaction is responsible to ensure that no more lock requests are needed before starting the shrinking phase, which is guaranteed by announcing a Commit to all the LMs that participate in the execution of the transaction.
- LM releases locks in bottom-up or leaf-to-root order (release locks on facts first, then on objects), which is the reverse of the top-down or root-to-leaf direction in which they were granted.
- In order to improve on the efficiency of executing Commit, the LM_i must release the locks on Read (q, Q) earlier than Writes or to release all locks after FM_i acknowledging the Commit request. This saves the overhead of one message at the expense of decreasing concurrency by holding Read locks a little longer than required. When the TM_j receives all Commit acknowledgements, it updates its log by signing < Complete T-id >. This is done to facilitate recoverability. The TM_j also sends *Complete* as the last request to the LM_i, for recovery purposes of RM_i.
- As far as the granularity is concerned, locking an object implies locking all the facts composing it. Thus locking is an inherited phenomenon in the hierarchy. Also, it requires that LM_i prevent two transactions from setting conflicting locks on two granules that overlap.
- In case of requesting to insert a new fact, the fact should be locked for read/delete. When a read/delete request is for a fact that has not yet been inserted (it might be locked in the FLT for that purpose,) then this request should be placed in the WFQ. If the request (read/delete) is for a fact that has not arrived yet, then the TM should be notified. The decision to continue is one of the TM's tasks.
- In order to prevent deadlock situations that may arise when several transactions create a cycle in the global Wait-For Graph [2], the ordering of transactions by their T-id's at that site is used. Thus, when T_j arrives and has a conflicting lock request with a T_k that is already locked, LM_i must decide to proceed according to the following rules:
  - If T_j-id > T_k-id then T_j will be inserted in the WFQ in the proper order of the waiting transactions and the TM_j will be acknowledged.
  - If T_j-id < T_k-id then LM_i must inform TM_j to rerun T_j-id with a new id.

  The second situation might arise due to a conflict between two TMs requesting the same exclusive lock on a fact/object at both the primary and the inverted sites. Each request of both TM_k and TM_j arrives at one site before the other. This situation is not likely to continue to happen after rerunning the late T_j, since one part of the

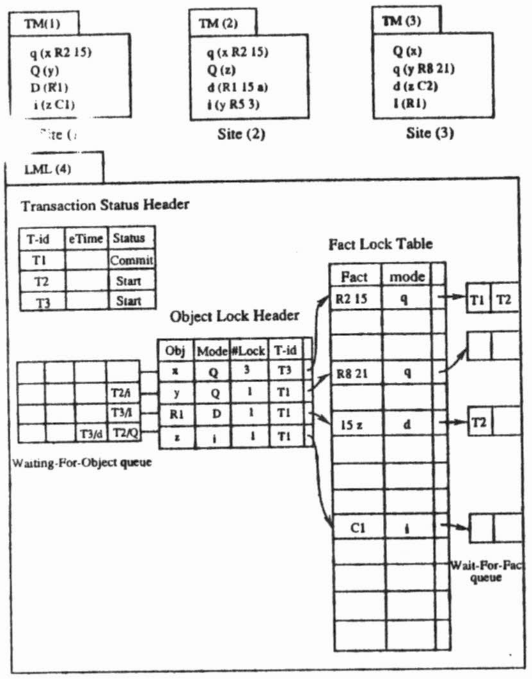


Fig (5) : A Snapshot of the Lock Management Tables of LM(4)

previous conflict, namely T_k-id is assumed to have been executed and since T_j gets a new T-id.

4.3 Recovery From Failure

Recovery Manager RM should have the duties of logging all the changes of the database state before they take place, in oreder to be able to retrieve any data lost when a failure occurs. Also, the RM is responsible for managing the process of putting the node on-line again and acknowledging other involved nodes. Therefore, each RM has two main tasks: one is while the node is active and the other is when the node comes to life after a failure. **Active Node Task:**

- When a Start(T_j) in received from LM_i, RM_i will record that on its log along with the time and T_j-id.
- When a delete/insert request is received, it is recorded on the log.
- An Abort/Commit is recorded before any acknowledgment to the LM.

Recovery From Failure Task:
The first step for the RM is to estimate how long the site was idle by comparing the time of the last operation on the log with the local clock. If the time elapsed has exceeded the predefined time (timeout period,) the RM must send a recovery message to all the sites that were involved before the incident. The RM checks its log from the last checkpoint (which has been marked before as a normal periodical routine in the active task) and proceed as follows:

- If there is Start command without a correspondent Commit/Abort, an Acknowledgement to TM_j is performed. (Assuming a previous message has been sent before, this can be tolerated in the TM routine).
- If the crash is before a Commit, while there have been delete/insert operations, then Undo is performed on these operations.
- If the crash is after a Commit and before a Complete, then a Redo is performed.
- If the crash is after a Complete then no action is taken.

The log operations should be idempotent, that is they can be performed several times and yeild the same result (in order to ensure the recovery from a failure while doing a previous recovery process).

4.4 The Global Clock

It appears from the previous requirement that synchronizing clocks at different sites together is an important process to guarantee that a slower running clock will cause all lock requests from that site to be aborted in case a conflict might occur at any site. On the other hand, a fast running clock will have to wait more than required in the ordered queue before it is granted a lock. This synchronization issue can be dealt with either one of the following techniques:

- The LM_i is responsible for updating its site's clock according to the latest lock request from any TM_j. The TM_i also can share the same clock resource by keeping updating it according the latest acknowledgement from any LM_j. Those sites that have no activities during a specific period of time may be run away from the global time. They will be synchronized with the system by their next interaction.
- The other more general solution is to synchronize the clocks periodically among all sites. This can be done by a special transaction called Time-Synchronizing Transaction that should be sent to all the directly connected sites. This method will cause the propagation of the fastest clock time to all the network without much expenses on the traffic of all sites. This also might have the problem of driving the whole network faster/slower according to the maximum/minimum time.

5 CONCLUSION

A concurrent locking algorithm has been designed for a distributed database environment based on the Semantic Binary Model (SBM). SBM is a fact-oriented representation of an information system. Each fact is inversely replicated for the purpose of query efficiency. The proposed locking algorithm can be applied to the Linear-throughput Semantic Database Machine (LSDM), a multi-processor, multi-disk database machine which offers massive parallelism. The algorithm is deadlock free since transactions are globally ordered, and it is also free from starvation since the transactions are partially ordered on each fact/object waiting queue. The algorithm takes into account the possibility of a link failure.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[2] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4), December 1987.

[3] H.F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, New York, 1986.

[4] M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft. *Operating Systems: Advanced Concepts.* The Benjamin/Cummings Pub. Co, Inc., 1987.

[5] M.K. Pillalamarri, S. Navathe, and A.C Papachristidis. Semantic Database Modeling: Survey and Research Issues. In *Conference on VLDB, Brighton, England, September 3-5*, 1987.

[6] N. Rishe. *Database Design Fundamentals: a Structured introduction to Databases and a structured Database Design Methodology.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] N. Rishe. *Database Design: The Semantic Modeling Approach.* Prentice-Hall, Englewood Cliffs, NJ, to appear in, 1990.

[8] N. Rishe. *Efficient Organization of Semantic Databases*, pages 114–127. Springer-Verlag, Lecture Notes in Computer Science,Vol. 367, 1989.

[9] N. Rishe. Semantic Database Management: from Microcomputers to Massively Parallel Database Machines. Keynote Paper, Proceedings of The Sixth Symposium on Microcomputer and Microprocessor Applications, Budapest, October 17-19, 1989.

[10] N. Rishe. Transaction-management System in a Fourth Generation Language for Semantic Databases. In H.H. Hamza, editor, *Mini and Micro Computers: From Micro to Supercomputers. Proc. of the ISMM Int'l Conf. on Mini and Microcomputers*, pages 92–95, Acta Press, 1988.