90-1C

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.

SPONSOR
The International Society for Mini and Microcomputers — ISMM
Technical Committee on Computers

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

An Implementation of Conservative Two-Phase-Locking for Parallel Semantic Database Machines *

Scott C. Graham *Naphtali Rishe*

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

Abstract

Massively parallel database machines achieve a high transaction throughput by allowing many transactions to access the database at one time. Concurrency control algorithms are used to insure that each of these parallel transactions works on a consistent database. This paper proposes an algorithm that adopts conservative two-phase-locking, a pessimistic concurrency control algorithm, to parallel semantic database machines. The conservative two-phase-locking algorithm actively avoids deadlocks, so no deadlock detection and recovery mechanisms are necessary.

The proposed algorithm is intended to run on a semantic database machine. The Linear-throughput Semantic Database Machine (LSDM) is an example of such a machine. The algorithm takes advantage of the LSDM's massive parallelism, easy expandability, high fault tolerance, and dynamic load balancing among processors without having to abort and restart transactions and without the overhead of deadlock detection and recovery.

The LSDM is optimized to store data using the Semantic Binary Model (SBM) of databases. The data is stored as elementary facts. These facts are ordered and replicated in a way that insures that almost all elementary queries of the database will require only one disk access. The proposed concurrency control algorithm will control access to the database by locking ranges of these facts; each range corresponds to an elementary query.

Keywords: concurrency control, database machines, semantic databases, parallel processing

1 Introduction

Massively parallel database machines achieve a high transaction throughput by allowing many transactions to access the database at one time. Concurrency control algorithms are used to insure that each of these parallel transactions works on a consistent database. Today, there are two schools of thought in concurrency control: optimism and pessimism. Each school has its advantages and disadvantages. Optimistic concurrency control algorithms allow each transaction to proceed under the assumption that few conflicts will occur. When a conflict does occur, one of the transactions must be aborted and restarted to allow the other to proceed. Pessimistic concurrency control algorithms lock some part of the database to prevent conflict. Since there are no conflicts, transaction aborts are not required. In most pessimistic concurrency control algorithms, however, deadlocks can occur. Optimistic concurrency control algorithms do not have deadlocks. This paper proposes an algorithm that adapts conservative two-phase-locking, a pessimistic concurrency control algorithm, to a parallel semantic database machine. Conservative two-phase-locking is deadlock free. No two transactions can be waiting for locks that the other holds, a requirement for deadlock, because the algorithm requires that all of a transaction's locks be available before any of its locks will be granted. The algorithm actively avoids deadlocks, so no deadlock detection and recovery mechanisms are necessary. When two transactions are waiting for the same lock, the first transaction that was submitted to the system will be given the first opportunity to hold the lock when it becomes available. No part of a transaction is run until all of the transaction's locks have been granted.

The proposed algorithm is intended to run on a semantic database machine. The Linear-throughput Semantic Database Machine (LSDM) [7] is an example of such a machine. The LSDM is a recent database machine model that offers massive parallelism, easy expandability, high fault tolerance, and dynamic load balancing among processors. The proposed algorithm will take advantage of the LSDM's features without having to abort and restart transactions and without the overhead of deadlock detection and recovery.

The LSDM is optimized to store data using the Semantic Binary Model (SBM) [3,4] of databases. The data is stored as elementary facts. These facts are ordered and replicated in a way that insures that almost all elementary queries of the database will require only one disk access [5]. The storage structure is partitioned between the storage units of the processors of the database machine [7]. The proposed concurrency control algorithm will control access to the database by locking ranges of these facts; each range corresponds to an elementary query.

Section 2 of this paper presents a brief introduction to concurrency control, a description of conservative two-phase-locking and a method for locking data in the semantic binary model. Section 3 proposes a pessimistic concurrency control algorithm and a description of how it can be used to implement concurrency control.

2 Concurrency Control

Concurrency control is used to make sure that each transaction in a database system operates on a consistent database for its entire life. It can perform this task by limiting access to data that is being used by another transaction or by aborting a transaction that has been caught working on an inconsistent database. In a parallel database machine, concurrency control requires a transaction to see the data contained at each node at one logical time. The data that has been read from a node can not be changed before we use that data to perform an update to the database, or inconsistencies may occur. The concurrency control algorithms for a parallel database can be kept at one central site or they can be distributed throughout the parallel database.

There are currently two main methods used for the concurrency control of distributed and parallel databases. Optimistic concurrency control [6] allows a transaction to have free access to all of the data contained in a database. When two transactions conflict over access to some data, one of the transactions must be aborted and restarted. Pessimistic concurrency control (usually implemented by two-phase-locking (2PL) [2]) requires locks to be placed on data. The locks usually come in two different flavors: shared locks (also known as read locks) which allow many transactions to access the locked data at once, and exclusive (or write) locks which allow only one transaction to access the locked data. Two locks that can be placed on the same data at the same time are called compatible locks. Read locks are compatible with other read locks; a write lock is not compatible with any other lock. The use of both read and write locks allows several read operations to work on the same part of the database at the same time while disallowing updates on this same data. If many read operations come into the system for some popular piece of data, they may starve transactions requesting write locks on the same data. This problem can be solved, with a loss of concurrency, by making all of the locks exclusive locks.

*This work has been supported in part by a grant from the Florida High Technology and Industry Council.

2.1 The Locking Model

The concurrency control protocol proposed in this paper is based on a variation of 2PL called conservative 2PL [1]. This protocol forces a transaction to obtain all of its required locks before it performs any operations on the database. This causes more complexity for the database programmer, because the transaction's read and write sets must be declared before the transaction begins. A transaction can not hold any locks if it can not hold all of the locks that it has requested.

The advantage gained from pre-declaring our read and write sets is a deadlock free locking protocol. From the definition of conservative 2PL it is obvious that if a transaction T_i is waiting for a lock held by transaction T_j, then T_i is holding no locks. This means that no other transaction T_k can be waiting for a lock that T_i holds. Therefore, there are no edges $T_k \rightarrow T_i$ in the wait-for graph (WFG) of the system. Since there are no such edges, T_i can not be in a WFG cycle, and thus can not be in a deadlock.

Conservative 2PL is a very pessimistic concurrency control scheme. It requires a transaction to maintain locks on its data the whole time that the transaction is executing. There can be no conflicts over data that can cause aborts, so each transaction will complete if it does not violate the database's integrity constraints. When a transaction is completed, it releases all of its locks, so no transaction will have to wait indefinitely to gain its locks. The implementation of conservative 2PL presented in this paper uses both read and write locks in order to increase concurrency.

2.2 The Locking Method

To maintain locks on data stored using the semantic binary model, we must decide exactly what kind of data we want to control and the method that we will use to control the access to this data. The database is composed of individual facts which are grouped by the data that they represent. All of the facts directly pertaining to a particular object are kept in one area of the database. Inverted facts are kept for each elementary fact. To place a lock on some data, we store the range of facts that is to be locked. Each range of facts corresponds to one elementary query. In order to place a write lock on data, the affected elementary and inverted facts must both be locked. If a requested lock's range intersects with some currently locked range, then the lock types must be compatible in order for the request to be granted.

Facts in the storage model [5] are represented by tuples representing relationships between the objects of the database. The locking ranges are represented by a starting tuple and an ending tuple. The tuple may be either an entire fact or some part of a fact. If the tuples S_s and S_e are the starting and ending tuples, respectively, of a lock range then any facts F_i in the database that satisfy $S_s \leq F_i \leq S_e$ are locked by that lock range. If a lock range spans two (or more) nodes of the system, the range should be broken into two or more smaller lock ranges that each fall onto one node.

Each node in the LSDM will maintain a list of the currently locked ranges of its data. This list will enable the node to determine whether a lock can be granted. The algorithm for the Lock Manager at each node is presented in Section 3.3.

The transaction does not access the data in any way until all of the locks that it needs have been granted. The algorithm that is used to gain the necessary locks is presented in Section 3.4.

3 An Algorithm to Implement Conservative 2PL

3.1 The Time Stamp / Transaction Id

Each transaction has a time stamp associated with it that acts as its transaction identifier (id). This time stamp is unique for each transaction. Each time stamp / id also represents the order that the transaction was submitted to the system. By examining two transactions' time stamps we can determine which transaction was submitted first; a priority can then be assigned to each transaction's lock request. A global system clock can be used to implement this time-stamping. A unique transaction-id can be formed by concatenating the global clock value with the site identifier of the transactions's site. One possible algorithm for maintaining the global clock is to synchronize adjacent sites periodically. Each site can send its current clock value to all sites which are connected directly to it. When a site receives a synchronization message, it will update its own clock value to the new time if it finds that its clock is slow. This will keep all of the sites' clocks advancing at the same rate and will prevent any site from gaining an advantage over the others by having a slow running clock. The terms transaction-id and time-stamp will be used interchangeably throughout this paper.

3.2 The Messages

In order to coordinate the locking and unlocking of data, several message types are defined. From the Lock Requester's viewpoint there are two general types of messages: those that it sends to the Lock Manager (outgoing messages) and those that it receives from the Lock Manager (incoming messages). Each of these types can be broken down as shown below. The messages all contain the message type, the lock range, the lock type, and the transaction id that the message pertains to.

There are two types of incoming messages.

- A *LockAvailable* message is received when a requested lock is available.
- A *WillCall* message means that the requested lock is currently unavailable, but the Lock Manager will send a *LockAvailable* message when it becomes available

There are four types of outgoing messages.

- A *LockRequest* message is sent when a transaction wants to lock a data range.
- An *Uninterested* message is sent to a Lock Manager when a transaction is not ready to lock a data range because it can not gain all of its locks.
- A *Re-interested* message is sent to a Lock Manager when a transaction believes that all of its locks should be available. It reactivates the lock request.
- An *UnlockRequest* message unlocks some data range that a transaction has locked.

3.3 The Lock Manager

Each node has a Lock Manager that maintains a list of the locks that have already been granted so that it can determine whether a lock request can be granted. If a lock request can not currently be granted, the Lock Manager must remember the request until it can be granted. The data structures and the algorithm for doing this are presented in Sections 3.3.1 and 3.3.2.

3.3.1 The Lock Manager's Data Structures

There are two major data structures kept at each node of the Lock Manager. One keeps track of the locks that have been granted, while the other holds requested locks that are currently unavailable and requested locks that transactions are not ready to impose.

- The *LockList* is a list of the locks that have been granted. This list is kept in the order of the lock ranges in order to make it very easy to compare a new request's lock range to those ranges that are already locked. It contains:
  1. the range that is locked
  2. the time stamp / id of the transaction
  3. the type of the lock
- The *RequestList* is a list of the locks that have been requested but not granted, as well as the locks that have been rejected, for the time being, by the requesting transaction. This list is ordered by the transactions' time stamps so that we can search through the list in the order that the transactions were generated whenever it is possible that a new lock may be granted. It contains:
  1. the range that has been requested
  2. the time stamp / id of the transaction
  3. the type of lock that was requested

4. a Boolean value, *interested*; if *interested* = TRUE, the locking transaction will be notified when the lock becomes available

3.3.2 The Lock Manager's Algorithm

The Lock Manager's algorithm is best described by listing the responses that it makes to each type of message that it receives.

- *LockRequest* — We should check to see whether the lock can be granted and respond accordingly.

```
/* check to see if the requested lock has
   already been given */
Locked := (Is any part of the range already
          in the LockList?)
LockCompatible := (Are all intersecting locks compa-
                  tible with the requested lock?)
if Locked then
  if LockCompatible then
    /* already locked but lock type is compatible */
    place the request on the LockList
    return a LockAvailable message
  else
    /* the requested lock is incompatible */
    place the request on the RequestList
          with interested = TRUE
    return WillCall message
else /* not locked */
  place the request on the LockList
  return a LockAvailable message
```

- *Uninterested* — We should put the lock request on the *RequestList*, marked so that we are not notified if its availability changes, and check to see if any entries in the *RequestList* can be granted.

```
remove the entry from the LockList
place the entry in the RequestList with
      interested = FALSE
treat each entry in the RequestList for which
      interested = TRUE and whose range intersects
      with the range just removed from the LockList
      as a LockRequest, but do not return a WillCall
      message if the lock is unavailable
```

- *Re-interested* — We should mark the lock request so that we will be notified when it becomes available and return the lock request's current status.

```
remove the entry from the RequestList and treat
      it as a LockRequest
```

- *UnlockRequest* — We should remove the lock from the *LockList* and check to see if any locks in the *RequestList* can be granted.

```
remove the entry from the LockList
treat each entry in the RequestList for which
      interested = TRUE and whose range intersects
      with the range just removed from the LockList
      as a LockRequest, but do not return a WillCall
      message if the lock is unavailable
```

3.4 The Lock Requester

Each transaction must place locks on all of the data that it will access. The data structures and the algorithm that are used for requesting locks are presented below.

3.4.1 The Lock Requester's Data Structures

There are two data structures that are used while requesting a transaction's locks. The first holds information about the transaction's locks. The second data item holds the transaction's time stamp / id.

- *Locks* is an array of the locks that the transaction needs, along with a status flag for each lock. Each array element contains:
  1. the lock range
  2. the lock type
  3. the lock status; it is TRUE if the lock is known to be available
- A time stamp / transaction id for the transaction; it should adhere to the guidelines set up in Section 3.1.

3.4.2 The Lock Requester's Algorithm

This is the algorithm that a transaction uses to get its locks.

```
NumWillCall := 0
NumLockAvailable := 0
NumLocks := the number of locks we are requesting
send out a LockRequest message for each lock range
for each lock that was requested do
    receive a message back from a Lock Manager
    if message = LockAvailable then
      NumLockAvailable := NumLockAvailable + 1
      mark the lock status as available
    else /* message = WillCall */
      NumWillCall := NumWillCall + 1
      mark the lock status as unavailable
end for loop
while NumWillCall > 0 do
      for each lock that was requested do
          if lock status = available then
            send Uninterested message
            mark the lock status as unavailable
      end for loop
      do the following NumWillCall times
          receive a message from a Lock Manager
          reply with an Uninterested message
      end do loop
      for each lock that was requested do
            send Re-interested message
        end for loop
        NumWillCall := 0
        NumLockAvailable := 0
        for each lock that was requested
            receive a message from a Lock Manager
            if message = LockAvailable then
              NumLockAvailable := NumLockAvailable + 1
              mark the lock status as available
            else /* message = WillCall */
              NumWillCall := NumWillCall + 1
              mark the lock status as unavailable
        end for loop
  end while loop

  /* the while loop has terminated; all of the lock requests
     are available and have already been granted...*/
```

3.5 Use of the algorithm

When a transaction comes into the system, the following algorithm is used to process it:

1. Determine the read and write sets; we can make the transaction's programmer declare them, or determine them from the data that was requested.
2. Run the Lock Requester algorithm.
3. At this point, all of the locks that were requested have been granted. Accumulate the sets to be inserted and deleted by the transaction as we normally would in the semantic binary model. The nodes do not have to check that locks have been granted, because a transaction will not attempt to accumulate the sets unless it has all of the locks that it needs.

4. Check the integrity constraints. If the transaction does not pass the integrity check, return an error message to the user and skip to step 6.

5. Perform the accumulated transaction.

6. Unlock all of the data ranges by sending an *UnlockRequest* message for each range.

4 Conclusion

Massively parallel database machines are capable of performing typical database operations with a high degree of concurrency, thus increasing the number of transactions that can be performed per unit time by the users of the database. This paper has presented a pessimistic method of concurrency control (conservative two-phase-locking) for a massively parallel database machine, namely the LSDM (Linear-throughput Semantic Database Machine). The algorithm is general enough to work on other parallel database machines, and on other semantic database models and their storage structures by changing the locking mechanism to one that would be appropriate for that semantic model and its storage structure.

Unlike other two-phase-locking (2PL) protocols, conservative 2PL forces the read and write sets of a transaction to be defined before the transaction can begin. Some implementations of optimistic models [6] allow query transactions to be performed while waiting for an update transaction to commit; conservative 2PL makes these queries wait for the updates to complete. The conservative two-phase-locking protocol offers advantages over other two-phase-locking protocols because it is deadlock free, and over optimistic concurrency control methods because it avoids aborts and roll-backs. The time gained by avoiding these aborts and roll-backs should make up for the slight loss of concurrency that occurs because the queries can not be made during an update transaction.

Conservative 2PL, the semantic binary database model, and the LSDM all combine to form an effective implementation of a massively parallel database machine. This database machine will exhibit a large degree of concurrency and a high total throughput of transactions.

References

[1] P.A. Bernstein. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[2] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locks in Data Base Systems," Communications of the ACM, v. 17.7, pp. 403-412.

[3] N. Rishe. *Database Design Fundamentals: A Structured Introduction to Databases and a Structured Database Design Methodology*. Prentice-Hall. Englewood Cliffs, NJ. 1988.

[4] 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, pp. 1-12.

[5] N. Rishe. "Efficient Organization of Semantic Databases," Foundations of Data Organization and Algorithms. Eds. W. Litwin, and H.J. Schek. Springer-Verlag Lecture Notes in Computer Science, Vol. 367, pp. 114-127, 1989.

[6] N. Rishe, D. Tal, and E. Gudes, "Concurrency Control Algorithms for Distributed Storage Semantic Database Machines," FIU SCS TR 89-003.

[7] N. Rishe, D. Tal, and Q. Li, "Architecture for a Massively Parallel Database Machine," *Microprocessing and Microprogramming* (the Euromicro journal), *25* (1989), pp. 33-38.