A Dynamic Granularity Locking Protocol
for Tree-structured Databases

S. Haldar and D. K. Subramanian
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012, INDIA

Abstract
A dynamic granularity locking protocol for tree-structured databases is presented. It is a variant of multi-granularity locking protocol which takes the system load condition and the conflict status of the transactions into an account while locking a data granule. It shares the advantages exhibited by both coarse and fine granularity locking protocols, and retains the power of multi-granularity locking protocol. It dynamically changes the granule size of the data to be locked depending upon both the transaction-requirement and the current system load. The strategy of the protocol is to lock coarse granules at light system load or when transaction conflicts are less, and to lock fine granules at heavy system load or when the conflicts are more. The protocol uses strict two phase locking, in conjunction, to ensure serializability. A simulation study has also been done to study the performance of the proposed protocol.

1 Introduction
A database is a collection of related data items operated on concurrently by a set of transactions [4, 5]. Each data item is assigned a value from a well defined domain. The values of the data items comprise the states of the database. A database state is said to be consistent if the values of the data items satisfy a set of integrity constraints of the database [5]. A database access is an execution of a read or write operation on a data item of the database.

A transaction is a sequence of database accesses generated at the execution of its master program. It is assumed that each transaction is correct, i.e., when executed alone on an initially consistent database state will terminate in a finite time, produce correct results, and leave the database in another consistent state. When many transactions operate on a database in an interleaved fashion, there is a chance of the database becoming inconsistent. The transaction management unit of the database system imposes the task of avoiding inconsistencies on a concurrency controller (popularly known as scheduler). The scheduler preserves the database consistency by producing serialized execution of concurrent transactions. The scheduler must not only ensure the preservation of the consistency of the database, but also allow a possible amount of the interleavings of the operations. Moreover it should be an efficient scheduler.

Concurrency control in database systems has been a major focus of research. There are hundreds of schedulers available for concurrency control in database systems. The design of most of the schedulers are based on locking [5, 6] or timestamp ordering [2, 16]. When an enormous amount of data is to be managed, it is sometimes advantageous to organize the data in a hierarchical tree-like structure, especially when the hierarchy of the data is to be preserved. This helps easy management of data. Consider a database of an educational institution. The database may be partitioned into divisions, a division into departments, a department into areas, an area into files, a file into records, and so on (cf. Figure 1). A database is called a tree database if the data items are organized as nodes of a tree.

We shall refer to a unit of database, which can be locked by a scheduler, as a granule. In reality, a data granule could be a block of disk, a file, a record of a file, a field of a record, etc. Granularity of a data item refers to the item's relative size. A data granule is referred to as coarser if the granule contains relatively more ingredients than a finer granule. For example, the granularity of a file is coarser, and the granularity of a record in that file is finer. The granularity of a data item is not an important issue as far as correctness is concerned (therefore the correctness of the proposed protocol is not given here), but it is an important issue for the performance of the system [4, p68].

Most of the commercial database systems (DBSs, in short) use some variants of two phase locking (2PL, in short) protocol, originally proposed in [5]. A lock based scheduler is also known as lock manager (LM, in short). An LM may also be used to lock different data granules for tree databases.

The size of the data granules varies from one database system to another. In some systems like R [1], a data granule may be as small as one record. Some systems

*Current Address: Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada, A1C 5S7

$01.00 © 1991 IEEE
like System 2000 [15] support a single granule covering the entire database. Using coarse granules incurs low lock maintenance overhead, as fewer locks to be managed; at the same time it reduces the level of concurrency, since operation conflicts are more frequent. \textit{(Two operations are in conflict if one of them is a write one).} For example, if a DBS allows file locking, the transactions that modify two different records in a file can not proceed concurrently. This leads to a poor response. Contrarily, using fine granules improves concurrency by allowing a transaction to lock only those data it requires. But, it requires more locks to be managed. This leads to high lock maintenance overhead.

The original conjecture by Ries and Stonebraker [13] was that coarse granularity locking (CGL, in short) would be preferred to fine granularity locking (FGL, in short). Again in [14], they have reported that a FGL protocol fares well if transactions are small and their database access patterns are random. In most of the DBSs, transactions are general longer (occasionally long transactions are encountered). Thus, FGL is preferred to CGL.

Selecting a data granule to be locked requires a proper balance between the locking-overhead and the amount of concurrency. A variant of 2PL protocol called multi-granularity locking protocol for tree-structured databases was introduced by Gray et al. [6, 7, 8, 9]. A multi-granularity locking (MGL, in short) protocol allows each transaction to use data granules most appropriate for its application. For example, if a transaction accesses many records in a file, it simply could lock the entire file than to lock all the required records. In general, a long transaction that accesses many data items can lock coarse granules, and a short one can lock fine granules.

In a DBS that uses MGL protocol, the granularity of a data item to be locked is completely determined by the transactions (application programmers). The system ensures locking exactly the same data granules as required by the transactions. Thus, the MGL protocol is static in nature, i.e., it does not take transaction load on a data granule and transaction conflicts into account when it locks that granule. Locking user specified data granules is advantageous during heavy system load or when conflicts are more, but has some apparent disadvantages during light load or when conflicts are less.

This paper has taken the above factors into consideration, and proposes a variant of MGL protocol called dynamic granularity locking (DGL, in short) protocol for tree-structured databases. The DGL protocol shares the advantages exhibited by both CGL and FGL, and also retains the power of MGL. A system following the DGL protocol allows transactions to use the appropriate data granules suitable for their applications, but it locks the most appropriate data granules depending on both transaction requirements and the current system load. The lock manager locks (tries to lock) coarse granules when the system is lightly loaded irrespective of the transaction requirements, and locks transaction-specific granules under heavy system load.

The performance of the DGL protocol has been evaluated and compared with those of CGL and FGL protocols for tree-structured databases using discrete event simulation technique. The results show that the DGL protocol is well suited for tree-structured databases.

Paper Layout

The rest of the paper is organized as follows. Section 2 briefly presents the MGL protocol. Section 3 presents the DGL protocol. A detailed simulation study is presented in Section 4. Section 5 concludes the paper.

2 Multi-granularity Locking Protocol

The hierarchical relationship between coarse granules and fine granules could be represented by a lock instance tree (cf. Figure 2, courtesy Figure 3.4 of [4]). The tree represents an abstract data structure to be used by the lock manager to maintain locks at different nodes of the tree. A lock (read or write) on a coarse granule explicitly locks it, and implicitly locks all of its descendants that are finer granules. For example, a read lock on an area data implicitly read-locks files and records in that area. It is also necessary to propagate the effect of locking a fine granule to those coarse granules that contain it. This is done with intention locks. The MGL protocol uses five types of locks. They are \textit{read lock (rl)}, \textit{write lock (wl)}, \textit{intention read lock (irl)}, \textit{intention write lock (iwl)}, and \textit{read intention write lock (riwl)}. \textit{rls} and \textit{wls} are used to explicitly lock different data granules. The intention locks are used to propagate the effect of locking a fine granule to those coarse granules that contain it. The \textit{rl}, \textit{wl}, and \textit{riwl} grant their holders the privilege of accessing some data granules. The intention locks \textit{irl} and \textit{iwl} do not have this property, but they serve as pre-requisite for the requesting of other locks. It is to be noted that each transaction always requests for \textit{read and write locks} on various data granules. It is the lock manager that uses intention locks for better concurrency control. The semantics of the locks are as follows [10]. Here \textit{T} denotes a transaction, and \textit{d} a data granule.

\textbf{Read Lock:} A \textit{rl} held by \textit{T} on \textit{d} allows \textit{T} to read any component of \textit{d} arbitrarily often, and to use their values in its computation as long as \textit{T} holds the \textit{rl} on \textit{d}.

\textbf{Write Lock:} A \textit{wl} held by \textit{T} on \textit{d} allows \textit{T} to read and/or modify any component of \textit{d} arbitrarily often, and to use their values in its computation as long as \textit{T} holds the \textit{wl} on \textit{d}.

\textbf{Intention Read (Write) Lock:} An \textit{irl} (iwl) held by \textit{T} on \textit{d} allows \textit{T} to set \textit{rl} (wl) or \textit{irl} (iwl) on any component of \textit{d}.

\textbf{Read Intention Write Lock:} Like \textit{rl}, a \textit{riwl} held by \textit{T} on \textit{d} allows \textit{T} to read any component of \textit{d} and to use their values in its computation as long as \textit{T} holds the \textit{riwl} on \textit{d}. Moreover, \textit{T} is allowed to set \textit{wls} or \textit{iwls} on any children of \textit{d}.
Lock Compatibility: Let T.1 and T.2 be two transactions, and X and Y two lock types. A lock of type X is said to be compatible with another lock of type Y if T.1 may acquire a X mode lock on a data granule d while T.2 holds Y mode lock on d. The compatibility matrix of the above mentioned lock types is given in Table 1 (courtesy Figure 13 of [10]).

2.1 The MGL Protocol

A lock manager, on behalf of a transaction T, following the MGL protocol, sets and resets locks according to the following rules. Here, G is a lock instance tree.

Rule 1: If d is the root of G, to read or write d, T must acquire a rl or wl on d.

Rule 2: To read (write) d, T must acquire a rl or wl or riwl (wl) on some ancestor of d.

Rule 3: If d is not the root of G, then to set a wl on d, T must have an iwl or riwl on d's parent.

Rule 4: If d is not the root of G, then to set a rl on d, T must have an iwl or riwl on d's parent.

Rule 5: To set an iwl on d, T must have an iwl or riwl on d's parent.

Rule 6: To set an iwl on d, T must have an iwl on d's parent.

Rule 7: The transaction T may not release an intention lock on a data granule d if it is currently holding locks on any children of d.

Rule 1 implies that if a transaction wants to use the entire database it must acquire an appropriate lock, rl or wl, on the root. Rule 2 states that a transaction, holding a rl (wl) on a data granule d, can access any finer granules that are descendants of d. Rules 3–6 state that lock-setting is root-to-leaf order, and Rule 7 states that lock-releasing is leaf-to-root order. Further details are available in [6, 9].

A MGL protocol prevents transactions from owning conflicting locks on the same data granule, but does not ensure serialized execution of concurrent transactions. To ensure serializability, the lock manager must follow 2PL protocol in conjunction with MGL protocol. The 2PL protocol defines when to set or release locks on data granules, and the MGL protocol defines how to set or release locks on or from data granules.

Lock Conversion: Let a transaction T hold X type lock on a data granule d. If T again requests a Y type lock on d, the lock manager must convert X type lock held by T on d into a lock type that is at least as strong as both X and Y. For example, if "X = rl" and "Y = iwl", the lock manager should convert the rl into a riwl. The lock conversion matrix for MGL protocol is given in Table 2 (courtesy Figure 3.6 of [4]). If T holds an "old lock type" lock on d and the current request is for "requested lock type" on d, then the table entry states the lock type into which the "old lock" should be converted.

3 The Dynamic Granularity Locking Protocol

The main motivation of the lock manager (DGLM) that follows the DGL protocol is to lock coarse granules, when ever possible, irrespective of the transaction-specified data granules. On failing to lock coarse granules, the DGLM opts for fine granules. In a nut shell, the locking part of the protocol is as follows. Let G be a tree database. The DGLM is currently processing a X type lock request on a data granule d in the database G. The DGLM starts locking data granules from the root of G down to d. It sets appropriate intention locks on data granules that are on the way to d starting from the root till a data granule d' available to lock in mode X or stronger than X. The DGLM sets locks in root-to-leaf fashion, and releases locks in leaf-to-root fashion.

Next sub-section defines the data structures used by the DGLM.

3.1 Data Structures and Function

The DGLM maintains a table called lock-table for the purpose of setting and releasing locks on different data granules. The DGLM inserts a tuple (T, d, X) in the lock-table to set a X type lock on a data granule d on behalf of a transaction T. Releasing the lock implies removing the tuple from the lock-table.

The DGLM maintains four variables (requested-data, request-type, current-data, and conflict) and a list (acquired-list) in its local workspace for each active transaction. The variable requested-data (RD, in short) indicates the currently requested data granule, request-type (RT) the current request type (read or write), current-data(CD) the data granule to be locked now, and the boolean variable conflict the conflict status of the current request. The acquired-list (AL) is a list of ordered pairs; the structure of an ordered pair is (d, X), where d is the requested data granule in mode X, and X' the granted data granule in mode X. The hosts X' is at least as strong as X. It is assumed that a variable (or list), say VAR ∈ {RD, RT, CD, conflict, AL} for a transaction T could be accessed by referring VAR(T).

Proper-child (PC) is a function of two arguments, (CD and RD), and returns the name of the data granule that is a child of CD and an ancestor of RD. For example, considering Figure 2, let CD = DB1 and RD = Ri1:1, then PC(CD, RD) returns A1.

The dynamic granularity locking protocol is described in the next sub-section.

3.2 The Protocol

Let the DGLM be currently processing an X type (read or write) lock request on a data granule d on behalf of a transaction T. The DGLM performs the following steps.

S1: /* data initialization */
CD(T) := root of the database tree;
RD(T) := d;
RT(T) := X;
conflict(T) := false; /*no conflict so far*/

S2: If RD(T) = CD(T) or not conflict(T) then
   /grant an X type lock /
   insert a <T, CD(T), X> in lock-table;
   insert a <<d,X>,<CD(T),X>> in AL(T);
   get and return the value of d to T;
   /move 1 level down the tree/
   goto S2
else/*grant intention type of X lock*/
   insert a <T,CD(T),iX> in lock-table;
   CD(T) := PC(CD(T),RD(T));
   conflict(T) := false; /*move 1 level down the tree*/
   goto S2 endif;

S3: if CD(T) is not locked then
   S3.1 'rl': if (X = read) then
      acknowledge the requester
   else /* X = write */
      if (CD(T) = RD(T)) or
         (no locks on CD(T)) then
         convert Y to rl, if possible;
         /convert read lock to a write lock/
         /or abort the transaction T*/
         /start here after activation of T*/
         let <<e,,>,<CD(T),Z>> in AL(T);
         convert Z to rl;
         acknowledge the requester;
   else /* convert rl to riwl */
      convert Y to riwl, if possible;
      /or abort the transaction T*/
      /start here after activation of T*/
      if the request was blocked then
         conflict(T) := true;
         convert Y to rl;
         /start here after activation of T*/
         let <<e,,>,<CD(T),Z>> in AL(T);
         convert Z to riwl;
         acknowledge the requester;
      else /* X = write */
         insert the request in a pending list;
         /start here after activation of T*/
         /suspend T's activities*/
         /start here after T's activation*/
         conflict(T) := true;
         goto S2;
   endif;
   /start here after activation of T*/
   if the request was blocked then
      conflict(T) := true;
      CD(T) := PC(CD(T),RD(T));
   goto S2;
endif;

S3.5 'iw1': begin
   CD(T) := PC(CD(T), RD(T));
   conflict(T) := false;
   goto S2;
end;
end {of case}

else /*T doesn't hold a lock on CD*/

S4: let Y be the strongest among the locks set on CD(T);
   case Y of
      S4.1 'wl': begin
         insert the request in a pending list;
         /suspend T's activities*/
         /start here after activation of T*/
         conflict(T) := true;
         goto S2;
      end;
      S4.2 'riw1': if (X = read) then
         insert a <T,CD(T),iX> in lock-table;
         CD(T) := PC(CD(T),RD(T));
         /move 1 level down the tree/
         conflict(T) := false;
         goto S2;
      else /* X = write */
         insert the request in a pending list;
         /suspend T's activities*/
         /start here after activation of T*/
         conflict(T) := true;
         goto S2;
      endif;
      S4.3 'r1': if (X = read) then
         insert a <T,CD(T),x> in lock-table;
         CD(T) := PC(CD(T),RD(T));
         /start here after activation of T*/
         conflict(T) := true;
         goto S2;
      endif;
      S4.4 'irl', 'iwl':
         begin
            insert a <T,CD(T),x> in lock-table;
            CD(T) := PC(CD(T),RD(T));
            conflict(T) := false;
            goto S2;
         endif;
      end {of case}
   endif {of if/else}
At the step S 1, the DGLM initializes the variables for the transaction T. At S 2.1, the DGLM locks CD if it is the required data or CD is currently free (not locked) and the current request has not conflicted with another one recently. At S 2.2, it grants an intention type lock, even if the data (CD) is currently free. It is to be noted that this step is executed only if the current request conflicted with another one recently. If T holds a lock on CD, the DGLM executes the step S 3. If T holds a rl on CD and the current request type is read, it grants the request immediately. At S 3.1.1, it converts a rl into a rl, if no other transactions hold any locks on CD. When other transactions hold some locks on CD, at S 3.1.2, it converts a rl into a rl. If T holds a rl, in CD, it grants the request immediately (S 3.2). S 3.3 suggests that if T holds a rl on a data granule d and the current request is on any descendant of d, the request should immediately be granted; otherwise the DGLM starts locking from one level down the tree. S 3.4 states that if T holds an rl on CD, and the current request type is read, the DGLM starts locking from one level down the tree. On the other hand, if the current request is a write one, the DGLM converts the rl to iwl, and starts locking one level down the tree. If T holds an iwl on CD, the DGLM converts the request into a riwl. If T holds a riwl on CD, it grants the request immediately (S 3.5). The complete step S 4 deals with the situation when T does not have any lock on CD, but other transactions hold some locks on CD. The steps S 4.1, 4.2.1 and 4.3.1 are executed, if the current request type is incompatible with the existing locks, to block the request. At steps S 4.2 and 4.4, the DGLM grants an intention lock, and at S 4.3 a read lock.

The above algorithmic steps define how to set locks on data granules. The lock release operation is exactly the same one followed by a MGL protocol, i.e., the locks are released in leaf-to-root order. To ensure serializability the DGLM must also follow 2PL protocol to determine when to set or release locks. The 2PL protocol also defines when to block or abort a request, and when to unblock a blocked request.

The lock compatibility matrix used by the DGLM is exactly the same as given in Table 1. The lock acquisition and conversion techniques mentioned in the protocol description are summarized in Tables 3 and 4.

4 Simulation Studies

To evaluate the performance and the effectiveness of the DGL protocol over MGL protocols an extensive simulation study is done. Some of the results are presented here. Various assumptions, that are made in the process of simulation, are presented below.

1. The model, considered here, is shown in Figure 3. The system, considered, has a single processor, infinite main memory, an on-line database storage device, and a log storage device. The database is assumed to be memory-resident with checkpoints and 'redo' logs [11, 12]. The database is assumed to be a (logical) complete binary tree. The leaf nodes (the finest granules) of the tree represent physical data, and the internal nodes represent logical data (the coarse granules). The data at the leaf level are of the same granularity, and are the unit of transfer between main memory and storage devices. It is assumed that there are 1024 finest data granules in the database.

2. The log and database storage devices considered here are fixed head magnetic disks. A disk-access is modeled by its latency and transfer time. The maximum latency time and data transfer time are assumed to be constant 20.0 msecs and 0.716 msecs, respectively. The disk-access time is uniformly distributed between 0.716 msecs and 20.716 msecs.

3. The transaction arrival pattern follows a Poisson distribution with a mean arrival rate λ. Transactions always access finest data granules. It is assumed that each transaction access five such data granules. All finest granules are accessed with the same probability. The transactions access data in a predetermined order to avoid deadlocks and premature aborts. The transactions do not perform any local computations. All the updates made by a transaction are written into the log with a single write.

4. The overhead assumed for lock set and release, transaction blocking and activation, etc. are summarized in Tables 5-8.

4.1 Results and Discussion

This section presents the results obtained from different simulation runs for various application environments and systems. The systems considered were - (1) a system that uses fine granularity locking protocol (FGLS), (2) a system that uses coarse granularity locking protocol (CGLS), and (3) a system that uses dynamic granularity locking protocol (DGLS). Three application environments considered were - (1) an environment characterized by 100% write operation, (2) an environment characterized by 80% write operations, and (3) an environment characterized by 20% write operations. The environments of all the systems are the same. The systems only differ in locking techniques. The FGLS always locks transaction-specified data granules starting from the root, and the CGLS locks logical data granules at the depth of three starting from the root.

Here, the average response time is considered as the performance measure. The response time of a transaction is defined as the time that elapses between its initiation and the reception of its response.

4.1.1 The Results

Scenario 1: Application Environment Characterized by 100% Write Operations.
The average response time characteristics for the assumed environment are shown in Figures 4 and 5. Figure 4 shows that at light system load, i.e., at low transaction arrival rate), the DGLS offers the best response time characteristic. For example, in DGLS, the average response time increases from 64.0 msecs at λ = 0.01 transaction/sec, to 122 msecs at λ = 10 transaction/sec. [Unless specified otherwise the unit of λ is transaction/sec]. However, the average response time for FGLS and CGLS increases from 73.8 msecs and 65.6 msecs respectively at λ = 0.01, to 126.6 msecs and 122.7 msecs respectively at λ = 10. Let us consider the point λ = 4 for performance comparison. At this point, the average response time for FGLS is approximately 76.0 msecs; it reduces to 67.3 msecs for CGLS; it further reduces to 60.8 msecs for DGLS, that is, the average response time of DGLS is approximately 86.6% that of FGLS, and 97.7% that of CGLS. Figure 4 clearly states that at light load, the response time characteristic for DGLS is closer to that for CGLS.

The average response time characteristics, at higher system load, are given in Figure 5. It is seen from the figure that the CGLS provides the poorest performance, and the FGLS the best. The characteristic for DGLS closely follows that for FGLS. The average response time for DGLS (CGLS and FGLS) increases from 122 msecs (122.7 msecs and 125.5 msecs, respectively) at λ = 10, to 2.522 secs (6.81 secs and 2.192 secs, respectively) at λ = 20. At λ = 20, the average response time for DGLS is approximately 37% that for CGLS and 116% that for FGLS. The higher average response time for DGLS, than FGLS, at higher system load is due to the fact that the DGLM always tries to set locks on coarse granules, whenever possible. This leads to relatively more access conflicts.

Scenario 2: Application Environment Characterized by 80% Write Operations.

Figures 6 and 7 show how the average response time varies for the assumed systems and the environment. It is seen form the Figure 6 that the average response time is the lowest in the case of DGLS, and the highest in the case of CGLS. The average response time for DGLS (CGLS and FGLS) increases from 53.1 msecs (54.66 msecs and 63 msecs, respectively) at λ = 0.01, to 97.16 msecs (97.85 msecs and 95 msecs, respectively) at λ = 11.1. At low transaction arrival rate, the average response time for DGLS is approximately 37% that for FGLS, and 96.4% that for CGLS. This figure clearly states that the DGL protocol follows the principles of CGL protocol when the system is lightly loaded, i.e., when the conflicts are less. Like Figure 5, Figure 7 also shows that at higher system load, i.e., when conflicts are frequent, the DGL protocol follows the principles of FGL protocol.

Scenario 3: Application Environment Characterized by 20% Write Operation.

The average response time characteristics for the assumed systems and the environment are given in Figure 8. It can be seen from the figure that the average response time is the lowest for DGLS. The average response time for DGLS (CGLS and FGLS) increases from 18.2 msecs (19.78 msecs and 28.06 msecs, respectively) at λ = 0.01, to 24.71 msecs (25.8 msecs and 32.11 msecs, respectively) at λ = 20. At λ = 20, the average response time for DGLS is approximately 76.96% that for FGLS and 96% that for CGLS. The Figure 8 clearly tells that when the conflicts are rare the DGL protocol follows the principles of CGL protocol.

5 Concluding Remarks

A dynamic granularity locking protocol for tree-structured databases has been presented. This protocol shares the advantages exhibited by both fine and coarse granularity locking protocols, and retains the power of multi-granularity locking protocol. The protocol dynamically changes the granule size of the data items to be locked depending on the transaction-requirements, the current system load condition and the conflict status of the transactions. The strategy of the protocol is to follow the principles of coarse granularity locking protocol at light system load or when conflicts are less, and to follow the principles of fine granularity locking protocol at heavy system load or when conflicts are more.

A detailed simulation study has been done to evaluate the performance of the dynamic granularity locking protocol, and to compare with that of coarse and fine granularity locking protocols. The simulation results show that the dynamic granularity protocol is more suitable for tree-structured databases. Nevertheless, a few shortcomings are associated with the dynamic granularity locking protocol. First, it requires more memory space to store various tables. Second, the algorithm is little bit complex. Third, when conflicts are frequent, the performance of the dynamic granularity locking is relatively poor than that of fine granularity locking protocol. The last one could be overcome by allowing a transaction to push down (de-escalation), if possible, the locks of other transactions towards leaf nodes to avoid a possible blocking.

References


Table 1: Lock Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>rl</th>
<th>wl</th>
<th>iwl</th>
<th>riwl</th>
</tr>
</thead>
<tbody>
<tr>
<td>rl</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>wl</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>iwl</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>riwl</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
</tr>
</tbody>
</table>

y: Compatible  
n: Incompatible

Table 2: Lock Conversion Matrix

"Old Lock Types"

<table>
<thead>
<tr>
<th></th>
<th>iwl</th>
<th>rl</th>
<th>riwl</th>
<th>wl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requested</td>
<td>iwl</td>
<td>iwl</td>
<td>iwl</td>
<td>riwl</td>
</tr>
</tbody>
</table>

lock type

<table>
<thead>
<tr>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
</tr>
</thead>
<tbody>
<tr>
<td>riwl</td>
<td>riwl</td>
<td>riwl</td>
<td>riwl</td>
<td>iwl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
</tr>
</thead>
<tbody>
<tr>
<td>riwl</td>
<td>riwl</td>
<td>riwl</td>
<td>riwl</td>
<td>iwl</td>
</tr>
</tbody>
</table>

Table 3: Locking in the Presence of Other Locks

"Existing Strongest Lock on a Non-Leaf Data Item"

Current request type

<table>
<thead>
<tr>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
<th>iwl</th>
<th>riwl</th>
</tr>
</thead>
<tbody>
<tr>
<td>iwl</td>
<td>iwl</td>
<td>iwl</td>
<td>iwl</td>
<td>iwl</td>
</tr>
</tbody>
</table>

nl: no lock present;  
+: end of locking;  
| X | possible blocking before setting a X mode lock; |

<table>
<thead>
<tr>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
</tr>
</thead>
<tbody>
<tr>
<td>rl</td>
<td>riwl</td>
<td>rl</td>
<td>riwl</td>
<td>rl</td>
</tr>
</tbody>
</table>

Table 4: Lock Conversion Matrix

"Old lock type"

<table>
<thead>
<tr>
<th>clinc2-7 request type</th>
<th>rl</th>
<th>riwl</th>
<th>rl</th>
<th>iwl</th>
<th>riwl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>iwl</td>
<td>iwl</td>
<td>iwl</td>
<td>iwl</td>
</tr>
</tbody>
</table>

- : no change of existing lock.

Table 5: Overhead (msec) to Lock a Free Data Granule

<table>
<thead>
<tr>
<th>rl(0.16)</th>
<th>wl(0.16)</th>
<th>iwl(0.126)</th>
<th>iwl(0.126)</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Locking Overhead (msec) in Presence of Other Locks

"Existing Strongest Lock on a Non-Leaf Data Item"

<table>
<thead>
<tr>
<th>C rl</th>
<th>B(0.09)</th>
<th>iwl(0.16)</th>
<th>iwl(0.16)</th>
<th>iwl(0.16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R rl</td>
<td>B(0.09)</td>
<td>iwl(0.16)</td>
<td>iwl(0.16)</td>
<td>B(0.09)</td>
</tr>
</tbody>
</table>

B: For blocking;  
CR: Current Request


Table 7: Lock Conversion Overhead (msec)

<table>
<thead>
<tr>
<th></th>
<th>rl</th>
<th>w1</th>
<th>lrl</th>
<th>w1l</th>
<th>r1l</th>
<th>w1r</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C&quot;</td>
<td>rl</td>
<td>-0.08</td>
<td>-0.08</td>
<td>-0.08</td>
<td>-0.08</td>
<td>-0.08</td>
</tr>
<tr>
<td>R</td>
<td>w1l</td>
<td>-0.08</td>
<td>-0.08</td>
<td>-0.08</td>
<td>-0.08</td>
<td>-0.08</td>
</tr>
<tr>
<td>T</td>
<td>r1l</td>
<td>0.12</td>
<td>w1l</td>
<td>0.12</td>
<td>b(0.09)</td>
<td>iwl(0.1)</td>
</tr>
<tr>
<td></td>
<td>riwl</td>
<td>0.073</td>
<td>-0.08</td>
<td>iwl(0.1)</td>
<td>-0.08</td>
<td>-0.08</td>
</tr>
</tbody>
</table>

A: Abort;
B: Block;
AO: Aborting Others and Blocking itself;
CRT: Current Request Type;

Table 8: Miscellaneous Overhead (msec)
Data structure setup time for each transaction = 0.12
Data structure reset time = 0.125
Lock release = 0.073
Unblocking a Pending request = 0.05

Figure 1: A Typical Tree-structured Database

Figure 2: A Typical Lock Instance Tree

Figure 3: A Typical Lock Instance Tree

Figure 4: Model of a Transaction Processing System Under Consideration

Figure 4: Average Response Time Characteristics