Inform. Systems Vol. 11, No. 4, pp. 275–286, 1986
Printed in Great Britain

0306-4379/86 $3.00 + 0.00
Pergamon Journals Ltd

PRIMARY COPY SYNCHRONIZATION FOR DB-SHARING

ERHARD RAHM
University of Kaiserslautern, Fachbereich Informatik, Postfach 3049, 6750 Kaiserslautern, West Germany

(*Received* 26 *September* 1985; *in revised form* 2 *April* 1986)

Abstract—In a database sharing (DB-Sharing) system multiple loosely or closely coupled processors share access to a single set of databases. Such systems promise better availability and linear growth of transaction throughput at equivalent response time compared to single processor database systems. The efficiency of a DB-Sharing system heavily depends on the synchronization technique used for maintaining consistency of the shared data. A promising algorithm is the primary copy approach which will be presented in this paper. We describe the actions of the lock manager in a basic and in a more advanced version. Furthermore, it is shown how the lock managers can be enabled to deal with the so-called buffer invalidation problem that results from the existence of a database buffer in each processor.

1. INTRODUCTION TO DB-SHARING

Many applications in online transaction processing as in banking, inventory control or flight reservation have a continually increasing need for high performance database management systems (DBMS). Such systems must chiefly fulfill the following demands:

—*High transaction rates*. Whereas current DBMS at best achieve about 200 (short) transactions per second (tps) of the DEBIT-CREDIT-type[1], high-volume applications require 1000 tps in the near future[2]. For 1000 tps a processor capacity upwards of 100 MIPS is needed, with more complex transactions such a processing capacity is necessary for far less tps.

—*High availability*. A typical requirement is an outage of five minutes per year only[2]. To provide sufficient fault tolerance each major hardware and software component should at least be duplicated[3], especially multiple processors are required. Furthermore, component failures should be transparent to the users, modifications in the software/hardware configuration must be performed online and the common database has to be a consistent and up-to-date reflection of the state of the business at any time.

—*Extendability*. The transaction system should allow to increase transaction throughput linearly by adding new processors. However, to keep the system acceptable for online-processing, response times must not increase significantly compared to single processor systems.

—*Manageability and maintainability*. The system must provide a high level interface to the end user and the programmer; in particular they should be relieved from the existence of multiple processors (single system image). Ease of installation, ease of maintenance and ease of modification are also very important because manageability and maintainability have direct influence to reliability. Recent investigations revealed that most system failures are caused by users and operators[4].

These requirements cannot be met appropriately by *tightly coupled* systems where all processors share a common memory and where only one copy of operating system (OS) and DBMS exists. Reliability and extendability are not sufficient since the common memory is a prime cause for system crashes and contention. Hence, the number of tightly coupled processors is typically low (four or less). In a *loosely coupled* system each processor has its own memory and a separate copy of OS and DBMS, and interprocessor communication is via messages only. Such systems offer the best framework for building a highly available system since they provide natural boundaries between the processors against unwanted interference[3]. Furthermore, they allow incremental expansion. However, loosely coupled multiprocessors may have performance problems since communication with messages is expensive in current OS, even if a high-speed communication system is used (see below). To make communication more efficient, one could use a common memory partition. Those *closely coupled* systems are considered in [5] and [6]; here, we discuss loosely coupled systems, which provide better availability and extendability.

A further aspect of classifying DBMS running on multiple processors is how the disks are connected to processors. There are two cases to distinguish:

—In *DB-Distribution* systems each processor owns some fraction of the disk devices and the database stored on them. Accesses to 'non-local' data require communication with the processor owning the corresponding database partition. This approach is used among others by the TANDEM NonStop system and many distributed database systems such as R*.

—In *DB-Sharing* systems each processor has direct access to the entire database. This implies that all

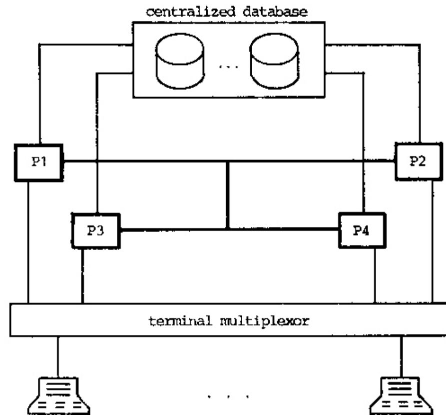


Fig. 1. Structure of a loosely coupled DB-Sharing system.

processors must be physically close (e.g. in one room) and permits a high-speed communication system (1–100 MB/sec). Examples of DB-Sharing systems are the Data Sharing facility of IMS/VS[7] and the AMOEBA project[8].

A detailed comparison between DB-Distribution and DB-Sharing can be found in [5] and [9]. Here, we concentrate ourselves on loosely coupled DB-Sharing systems as depicted in Fig. 1. The shown terminal multiplexor distributes each transaction entered on a terminal to one of the processors. A transaction can be completely executed within one processor since each processor has direct access to each data item of the centralized database. This capability avoids the necessity of a distributed 2-phase-commit protocol as required in DB-Distribution systems.

A main advantage of a DB-Sharing system is flexibility. Since each processor can access the entire database, transaction work may be dynamically distributed among the processors according to current needs and system availability. Additional processors can be added without changing the transaction programs or the database schema. Likewise, the failure of a processor does not prevent the surviving processors from accessing the disks or the terminals. Transactions in progress at a failed processor can be rolled back and redistributed automatically among the available processors.

To take advantage of the DB-Sharing architecture, new problems must be solved at first for the functional components depicted in Fig. 2:

—The *synchronization* component coordinates access to the centralized database. Since there is no common memory, synchronization requires message exchange among the processors which is much more time consuming than lock request handling in a centralized DBMS (e.g. a few hundred instructions for each lock grant or release). If the transactions synchronously wait for the response of a synchronization message, the waiting times directly influence the response time. Furthermore, it is necessary to

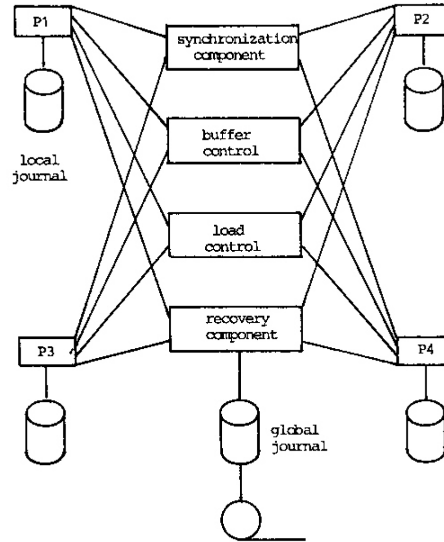


Fig. 2. Functional components of a DB-Sharing system.

bundle messages for transmission since sending and receiving messages are expensive operations at least in conventional mainframe OS. This delay of message collection also increases the transaction's response time just as the overhead for process or task switches due to synchronous transaction waits. To maintain throughput under these conditions, the level of multiprogramming in each processor has to be increased what, in turn, enlarges the conflict probability and OS overhead for scheduling, paging, etc. Therefore, high performance and acceptable response times are only reachable if the synchronization algorithm minimizes the average number of synchronous messages per transaction.

—*Buffer control* is needed to manage the problem of *buffer invalidation* that results from the existence of a local database buffer at each processor. An update only modifies the processor's local copy of a database object; copies in the buffers of other processors are getting obsolete. Access to invalidated objects has to be avoided and a method to propagate the new contents of modified objects to other processors must be supplied. If an update transaction writes the modified objects to the database before or during commitment (FORCE-strategy[10]) the latest version of an object can always be read from disk. Using a NOFORCE-strategy, modified objects may be exchanged directly via the inter-processor connections or also across the shared disks. In closely coupled DB-Sharing systems a common memory partition can also be used to exchange modified data[6].

—*Load control* has to find an effective distribution of the current workload against the set of available processors (transaction routing). No processor must be overloaded and the routing should support synchronization with minimal communication cost. To fulfill these tasks load control must react dynamically

to changes in the workload or within the system configuration (e.g. crash or reintegration of a processor). Transaction routing is usually done by analyzing the transaction type and possibly the input data. At least for short transactions this information is sufficiently precise to give a prediction of the presumable reference behavior of a transaction. More details about this subject can be found in [11, 12].

—The *recovery component* is responsible for system-wide logging and recovery. In addition to the local journals of each processor, there exists a global journal (e.g. for media failure) that can be constructed by merging the local log data. The recovery component has to ensure that, after a processor crash, the surviving CPUs recover and reintegrate the failed processor in order to continue transaction processing. Uncommitted transactions of the crashed processor are rolled back and restarted at another processor in order to preserve failure transparency to the user.

In this work we present solutions to the synchronization problem and to the buffer invalidation problem. Sections 2 and 3 describe how synchronization is performed with the primary copy algorithm in a basic and in an optimized version, respectively. After that several techniques to cope with buffer invalidation are given, all applicable in combination with primary copy locking. Synchronization protocols for DB-Sharing including optimistic methods are also considered in [13–16].

The primary copy algorithm to be described may not be confused with similar algorithms used in distributed database systems with replication[17]. In the latter schemes each data item has a primary copy controlled by one of the processors. Besides of synchronization, the primary copy processor is also responsible for updating all copies of the data item. With DB-Sharing, however, no replicated data must be controlled in this way (although another kind of replication is treated by the buffer control).

2. PRIMARY COPY SYNCHRONIZATION

As pointed out in the previous section, the major goal of a synchronization protocol in a DB-Sharing system must be the minimization of synchronous messages between the processors. This means that synchronization requests should be treated locally as much as possible.

An obvious protocol is the use of a *central lock manager* (CLM) for global synchronization. In such a scheme, the CLM resides at one processor and maintains a global lock table to answer lock requests from other CPUs. In the simplest form, each lock request is sent to the central lock manager for immediate processing. Since this straightforward strategy is certainly too expensive, the communication overhead has to be decreased e.g. by applying hierarchical locks. In such a scheme, the central lock manager shares the work with local lock managers located at each processor. Hence, a lock request can be handled locally if the local lock manager holds a (hierarchical) lock for the requested object. Local lock management is always possible if 'sole interest' exists for an object, that is only local transactions are interested in accessing the object. The usefulness of the concept of sole interest heavily depends on the locality of reference, which should be preserved by the load balancing algorithm as far as possible. However, in general, sole interest can be maintained only if a relatively small number of transactions references the object. Moreover, sole interest may be destroyed by a single stray reference from any other processor. Another shortcoming of this approach results from the CLM being a single point of failure.

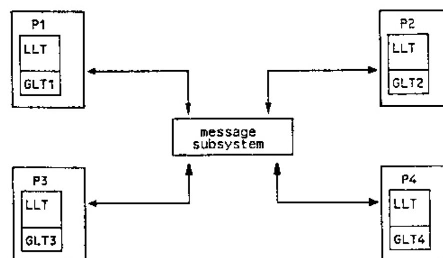
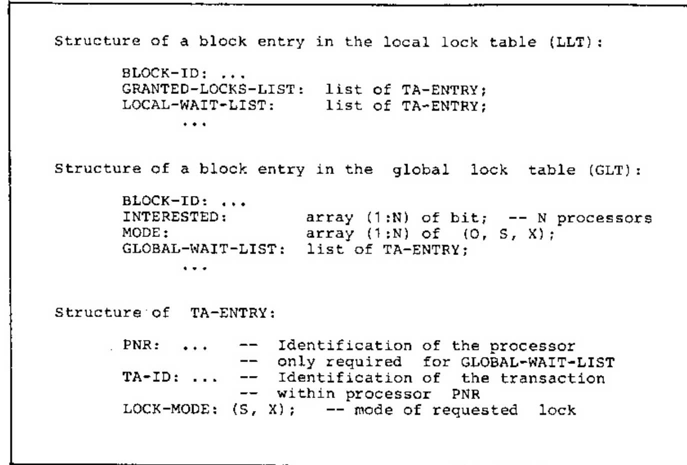


Fig. 3. Primary Copy Locking (N = 4).

A much more promising algorithm is *primary copy locking* (PCL), which is an extension of the CLM scheme in order to reduce the amount of lock request messages. Instead of having one CLM, the synchronization responsibility is now distributed among all processors. Therefore, the database is logically partitioned into N disjoint parts and each of the N processors performs the global synchronization for one partition. A processor is said to have the *primary copy authority* (PCA) for its partition. As Fig. 3 shows, each lock manager has a global lock table (GLT) to control the objects of its partition. As opposed to the GLT, a local lock table (LLT) keeps information about granted or requested locks for local transactions only.

Primary copy locking has the obvious advantage that lock requests from processor P within the partition controlled by P can be managed locally, regardless of external contention. Lock requests for a partition of another processor are sent to the authorized processor.

To take full advantage of the primary copy approach, transactions should not be routed to processors at random. Rather, the load control should attune the partitioning of the data and the assignment of the load such that the total number of "long" lock requests is minimal. Furthermore, the distribution of the PCAs and the routing strategy can be dynamically modified if a processor fails or is added, or if the transaction load profile changes significantly. Thus, the primary copy scheme allows a tight and effective



```
Structure of a block entry in the local lock table (LLT):

      BLOCK-ID: ...
      GRANTED-LOCKS-LIST:  list of TA-ENTRY;
      LOCAL-WAIT-LIST:     list of TA-ENTRY;
           ...

Structure of a block entry in the global lock table (GLT):

      BLOCK-ID: ...
      INTERESTED:        array (1:N) of bit;  -- N processors
      MODE:              array (1:N) of  (O, S, X);
      GLOBAL-WAIT-LIST:  list of TA-ENTRY;
           ...

Structure of  TA-ENTRY:

      PNR:  ...    --  Identification of the processor
                   --  only required  for GLOBAL-WAIT-LIST
      TA-ID: ...   --  Identification of  the transaction
                   --  within processor  PNR
      LOCK-MODE: (S, X);   -- mode of requested  lock
```

Fig. 4. Structure of the block entries.

cooperation with the load control permitting flexible adaptation to changing working conditions. These properties should result in much less messages for synchronization than using a central lock manager.

Algorithms for determining the PCA distribution as well as a strategy for transaction routing are presented in [12]. The essential input parameters of these algorithms are an appropriate description of the load profile and the number of processors.

In this and the next section we mainly describe the data structures required for synchronization and the actions of the lock managers to process a LOCK operation issued by a transaction. The remaining parts of the lock protocol (processing of an UNLOCK operation, structure and processing of the required message types) cannot be explained in full detail due to space limitations. Nevertheless, the provided information should give a notion of their possible realization. Throughout this paper we assume that synchronization is performed on block level (page level) and that two types of locks are obtainable for a transaction: read or shared locks (S-locks) and write or exclusive locks (X-locks). The compatibility of these locks is as usual.

Data structures

Both types of lock tables (local and global lock table) use a different format for their control blocks or block entries required for synchronization. The exact layouts of the block entries are given in Fig. 4. It is assumed that a block entry for a certain block may simultaneously reside in the global lock table of a processor as well as in the local lock table of this processor.

The example situation of Fig. 5 shows the block entries for a block B1 within the lock tables of two

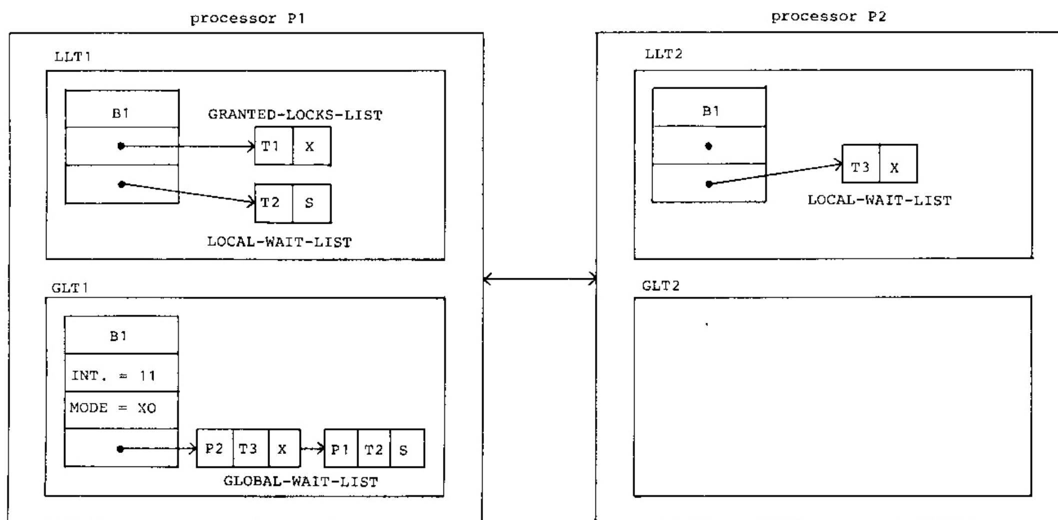


Fig. 5. Block entries in the lock tables (example).

processors. We assume that processor P1 holds the PCA for the partition to which B1 belongs. In the block entries of the LLTs only local transactions are kept, either in the list of the granted locks or in the LOCAL-WAIT-LIST. In the GLT, transactions waiting for a lock are stored within the GLOBAL-WAIT-LIST where transactions of all processors can wait.

In addition to the GLOBAL-WAIT-LIST, a block entry in the GLT contains further information (as seen in Fig. 4) in order to process lock requests. The vector INTERESTED indicates the processors that keep a block entry for the respective block in their LLT. The vector MODE gives the mode of granted locks for the interested processors. Value MODE (P) = 0 says that processor P is not interested in the block or that there are only transactions waiting for a lock on the block. Value MODE (P) = X indicates that an X-lock has been granted to a transaction at P, value S means that a S-lock was granted. In Fig. 5, 'INT. = 11' is used as an abbreviation of INTERESTED (1) = 1 and INTERESTED (2) = 1. This means that both processors are interested in the block. Similarly, MODE = X0 stands for MODE (1) = X and MODE (2) = 0. This says that an X-lock was granted to a transaction at P1 and that no lock for B1 is granted at P2.

Now we are in the position to explain lock request handling using the data structures just introduced. The sketched algorithm can be seen as a basic version of primary copy locking. In Section 3 we discuss an improved version.

Lock request handling

Assume transaction T at processor P has issued a lock request for block B. The basic version of PCL handles this lock request as follows:

If P is the processor owning the primary copy authority for the requested block, it is checked whether or not the GLT already contains a block entry for B. If this block entry does not exist, the lock can be granted since T is the only transaction that wants to access B. In this case, block entries for B are created within the GLT and within the LLT at P, the vectors INTERESTED and MODE are initialized properly, and T is inserted into the GRANTED-LOCKS-LIST of the block entry in the local lock table. If the GLT already holds a block entry for B, the lock request of T can be satisfied if the GLOBAL-WAIT-LIST is empty and if the required lock mode is compatible with the granted locks (decidable by using vector MODE). Otherwise, T has to wait for the desired lock and is appended to the GLOBAL-WAIT-LIST and to the LOCAL-WAIT-LIST in the LLT.

If P does not own the primary copy authority for block B, the lock request cannot be treated locally. So, T is entered in the LOCAL-WAIT-LIST in the block entry of B in the LLT (the block entry may have to be created at first) and a lock request message is sent to the responsible processor, say P′. P′ uses its GLT for processing the lock request message as just described for the local case. Only if the lock is grantable a lock response message is sent. Otherwise, T is appended to the GLOBAL-WAIT-LIST in the GLT of P′. In that case, the lock manager of P′ activates T (using a lock response message) at the point in time when the conflicting transactions have released their locks on B and T is chosen from the GLOBAL-WAIT-LIST to obtain the requested lock. After receipt of the lock response message, T is removed from the LOCAL-WAIT-LIST and inserted into the GRANTED-LOCKS-LIST.

For illustration, look at Fig. 5 once more. Assume that at the time when transaction T1 had issued its X-request for block B1, there was no interest in B1 at processor P2. Therefore, the GLT at P1 had contained INTERESTED = 10 and MODE = X0 for B1 when the X-lock was granted to T1. After that, suppose transaction T3 at P2 has wanted to modify block B1. In the LLT of P2 a block entry for B1 was created, then T3 was inserted into the LOCAL-WAIT-LIST, and finally a lock request message was sent to P1. This message resulted in the change of INTERESTED to 11 and T3 was inserted into the GLOBAL-WAIT-LIST since the requested X-lock was not compatible with the granted X-lock at P1. Figure 5 shows the situation after a lock request from transaction T2 issued at P1. This S-request of T2 was also appended to the GLOBAL-WAIT-LIST.

3. OPTIMIZATIONS OF THE PRIMARY COPY ALGORITHM

Unfortunately, the primary copy algorithm as described so far has the potential problem that it works well for a limited number of processors only, since its quality heavily depends on partitioning and load control. When a processor is added to the system the PCAs have to be redistributed. This, however, increases the probability that a transaction accesses data not controlled by the local lock manager. So, the number of lock request messages per transaction may grow with the number of processors if the new partitioning and transaction routing strategy cannot assure the same locality of references. This makes it difficult to achieve linear growth in transaction throughput (without increasing response times unacceptably) when adding new processors.

In order to reduce the dependencies to partitioning and load control, we now give an optimization that should often allow local lock management even for objects for which another processor holds the PCA. The improvement mainly consists of a mechanism that makes use of locality of reading references. A fast handling of S-requests is very important, because, in general, they are more frequent than X-locks (even in update transactions). The optimization of S-requests is especially interesting for level-2-consistency [18] being usual in existing DBMS. With level-2-consistency, S-locks are not held until EOT

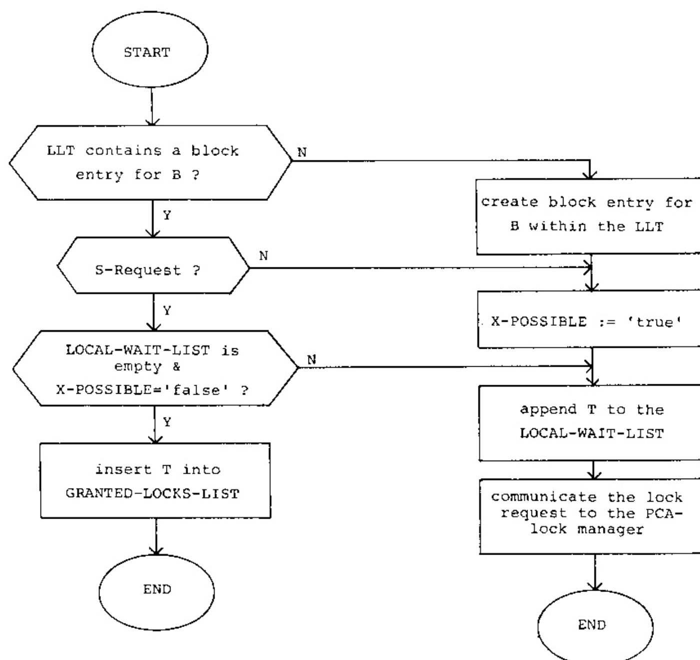


Fig. 6. Use of the LLT to process a LOCK operation on block B issued by transaction T.

but are short. So, a communication delay in order to grant a S-lock would last considerably longer than the time the lock is held. This would cause a significant increase in response time, especially in comparison with centralized DBMS.

Optimized S-request processing

To reduce the number of lock request messages for S-locks, we distinguish between two synchronization states for each block B:

State 1: An X-lock for block B is requested by at least one transaction in the system. The X-lock may be granted or not.

State 2: Block B is not in synchronization state 1. That means there is no interest in B or there are only reading accesses.

In synchronization state 1 the complete protocol as described in the previous section has to take place. So, each lock must be granted by the lock manager owning the PCA (of course, communication is only required if this lock manager does not reside at the transaction's processor). In particular, each X-request must be directed to the PCA-lock manager. However, in synchronization state 2 communication may be saved if each processor—irrespective of whether or not it holds the PCA—is allowed to grant S-locks. For this we add the following field in each block entry of the LLT that should indicate which synchronization state is given:

```
X-POSSIBLE: boolean;
                    (* default value: "true" *)
```

The value "true" ("false") of X-POSSIBLE corresponds to synchronization state 1 (2). If X-POSSIBLE has value "true" or if no block entry exists in the LLT, a lock request must be granted by the PCA-lock manager. If X-POSSIBLE has value "false" than S-locks can be granted locally. X-requests must always be directed to the PCA-lock manager. These cases are clarified by Fig. 6, where the actions of the lock manager are shown in order to process a LOCK operation of a local transaction using the LLT. If the lock request must be treated by the PCA-lock manager, the requesting transaction is inserted into the LOCAL-WAIT-LIST. When the lock is grantable, the PCA-lock manager issues a lock response that also contains information if X-POSSIBLE can be set to "false".

For a block B for which another processor holds the PCA, S-locks are only locally grantable if a block entry already exists in the LLT and X-POSSIBLE has value "false". To increase the probability of this case, block entries with X-POSSIBLE = "false" are not removed from the LLT when the last S-lock of a local transaction is released. This results in block entries with empty GRANTED-LOCKS-LIST and empty LOCAL-WAIT-LIST allowing that subsequent S-requests can be granted immediately. Therefore, locality of reading references within one processor is strongly supported independently of the PCA distribution.

A further optimization is that the release of the last S-lock need not be communicated to the PCA-lock manager when X-POSSIBLE has value "false" since the synchronization state remains unchanged. So, the

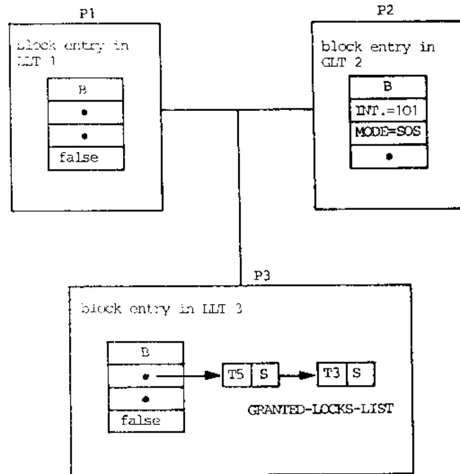


Fig. 7. Lock scenario in the improved PCL-scheme.

improved algorithm saves communication for requesting S-locks as well as for releasing it. As a consequence of not always notifying the release of the last S-lock, the value MODE (P) = S in a block entry of the GLT does not imply that S-locks for the block are actually granted at P but only that this is possible.

In the scenario of Fig. 7, it is assumed that P2 owns the PCA for block B. In the block entry for B in GLT2, processors P1 and P3 are kept as "interested" in B both with MODE = S. In the block entries of B in LLT1 and LLT3 the bits X-POSSIBLE have value "false". This gives P1 and P3 the right to immediately grant S-locks for B and indicates that the release of the last S-lock need not be notified to P2.

Processing of X-requests

The local handling of S-requests by processors not owning the PCA is only feasible if no X-requests for the block are in the system (synchronization state 2). If a transaction wants to modify a block in a situation like in Fig. 7, "empty" block entries in the LLT's (as at P1) have to be removed. For block entries with non-empty GRANTED-LOCKS-LIST (as at P3), the bit X-POSSIBLE is set to 'true' indicating that the release of the last S-lock must be notified to the PCA-processor. This is necessary for level-3-consistency in order to avoid that the page is modified at one processor and concurrently read at another. For level-2-consistency, a more effective handling of X-requests is possible (see below).

In Fig. 6 the processing of a lock request issued by a transaction T was only described with respect to the LLT. If the lock request cannot be granted using the LLT (what is always the case for X-requests), then the PCA-lock manager must further process the request using its GLT. The actions of the PCA-lock manager in order to process such a lock request are given in Fig. 8. The box LOCK-RESPONSE in this flow chart indicates that the lock can be granted to the requesting transaction T. If T is waiting at the PCA-processor, it can be continued immediately, otherwise a lock response message is sent.

In Fig. 8 the processing of an X-request has been left open for the situation where the GLOBAL-WAIT-LIST is empty and where no X-lock is granted. In that situation one or more processors J are kept with MODE (J) = S; these processors have a block entry with X-POSSIBLE = "false" for the requested block in their LLTs in order to grant S-locks locally. Since the X-request in that situation means a change of the synchronization state from 2 to 1, all these processors must be notified by using so-called STATE-CHANGED messages. The processing of the X-request and these STATE-CHANGED messages depends on the level of consistency that should be supported.

For *level-3-consistency*, the X-request on block B of transaction T is inserted into the GLOBAL-WAIT-LIST at first. Then, all processors J ≠ P with MODE (J) = S are informed about the new synchronization state by STATE-CHANGED messages. After receipt of such a message two alternatives are possible at a processor J:

(a) if no S-locks are locally granted at J ("empty" block entry), the block entry for B is removed from the LLT and this is communicated to the PCA-lock manager having the effect that INTERESTED (J) and MODE (J) are set to 0.
(b) if S-locks are currently granted at J, the bit X-POSSIBLE is switched to "true". The PCA-lock manager is getting informed after the release of the last local S-Lock on B.

The PCA-lock manager can grant the X-lock of T when it has received all responses to the STATE-CHANGED messages for after that it is guaranteed that no transaction is accessing block B.

Although this procedure ensures level-3-consistency, it makes clear that the optimized S-request handling considerably favours S-locks instead of X-locks. It may happen that X-requests must be delayed even when no S-locks are granted at any processor since several "empty" block entries have to be removed at first. Furthermore, the position of the PCA-processor is weakened since such a delay would be necessary even when the X-request has been issued within the PCA-processor (with basic PCL the X-request would have been granted immediately).

These shortcomings can be avoided when focussing on *level-2-consistency*. For this level of consistency, an X-request need not be delayed until all responses to STATE-CHANGED messages are arrived at the PCA-lock manager. Here, the X-lock from processor P can be granted immediately provided that no transaction at P is still owning a S-lock on the requested block.

The message STATE-CHANGED is sent to all processors J ≠ P with MODE (J) = S. Here, such a message has the effect that the block entry for block

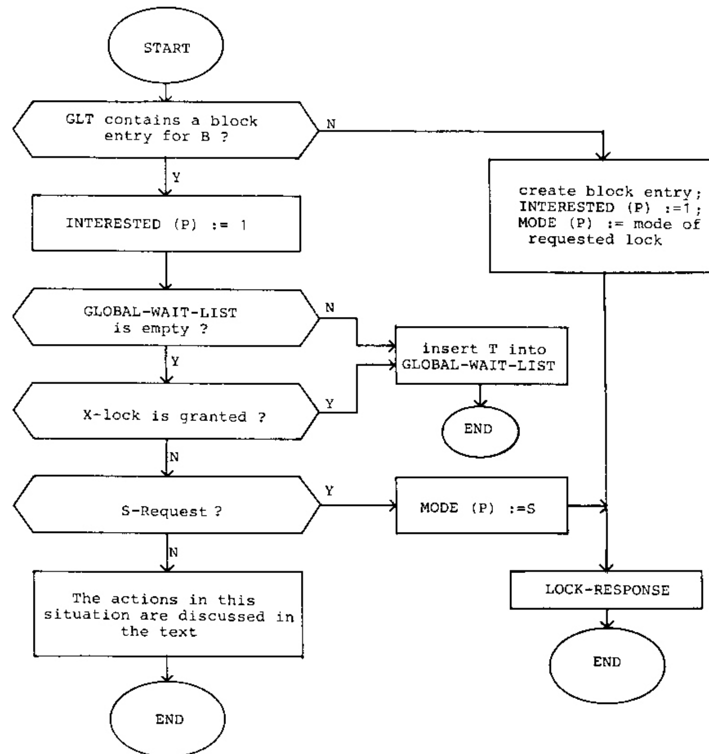


Fig. 8. Use of the GLT to process a LOCK operation on block B issued by transaction T running on processor P.

B is always removed from the LLT in the receiving processor—irrespective of whether or not S-locks are locally granted! This is feasible with the convention that a S-UNLOCK requires no treatment if the corresponding block entry cannot be found in the LLT. Removing the block entry from the LLT guarantees that no S-locks for B are (locally) granted after receipt of the STATE-CHANGED message. S-locks granted before the receipt of such a message only allow the reading of block B in its state before the modification by transaction T. This, however, does not affect level-2-consistency since no dirty data is read (note that unrepeatable reads are tolerated which occur when a transaction reads different versions of the same block).

So, an X-request can nearly always be granted immediately in the above mentioned situation where the X-request causes a change of the synchronization state from 2 to 1. Only when local transactions still hold S-locks on the requested block, the X-request must be delayed until the last of these S-locks is released in order to avoid that a local transaction reads dirty data.

This treatment of X-requests ensures level-2-consistency and prevents that the optimized S-request processing is at the expense of X-requests. It is supposed that the improvements will reduce the communication overhead for requesting and releasing S-locks to a minimum since locality of reading references is heavily exploited.

Activation of waiting transactions

The activation of a waiting transaction is usually done by the PCA-lock manager. This takes place after the release of an X-lock is notified or the release of all granted S-locks is ensured. The GLOBAL-WAIT-LIST need not be processed in FIFO-order, but more flexible strategies can be adopted in order to reduce communication, however, without starving some transactions. For example, it may be advisable to tell in a lock response message that two transactions in the receiving processor can obtain an X-lock one after the other. So only the last unlock must be notified. Similarly, one could grant all waiting S-requests of a LOCAL-WAIT-LIST when this is permitted for one of them. Alternatively, all these S-locks may be granted before an X-request is satisfied.

Deadlocks

So far we have said nothing about recognition and treatment of deadlocks. Deadlocks between local transactions can be treated as in centralized DBMS [e.g. if a lock request of transaction T cannot be satisfied it is checked if the delay of T causes a local deadlock; if so, the deadlock is resolved by aborting

one (e.g. the originator T) or more transactions]. For global deadlock management, the same techniques can be applied as for distributed database systems (conceivable techniques are described in [17, 19, 20]). The simplest solution to global deadlocks would be a timeout mechanism (a transaction is aborted after a fixed maximal waiting time due to external transactions); such an approach may be sufficient in many cases, because it is assumed that most lock requests can be done locally with primary copy locking. Therefore, global deadlocks should be a rare event.

Recovery aspects

To provide high availability, the concurrency control algorithm must be capable of correctly continuing synchronization after a processor crash. For the primary copy approach, the partition of a failed processor cannot be accessed until the crash recovery has been finished. In order to continue concurrency control for this partition, it is necessary to reconstruct the global lock table that was lost due to the processor crash. As explained in [21] this can be done by merging the block entries belonging to the partition to be recovered that can be found in the LLTs of the surviving processors. In the mentioned paper, there is also a description of how a new PCA distribution can be established.

4. SOLUTIONS TO THE BUFFER INVALIDATION PROBLEM

In this section, we describe some possible strategies to cope with buffer invalidations in the context of primary copy locking. We distinguish between systems with FORCE and NOFORCE propagation. FORCE-propagation requires that modified blocks of an update transaction have to be forced to disk before the transaction commits. This allows simpler solutions to buffer invalidation since the valid copy of a block can always be read from disk. With NOFORCE-propagation the blocks on disk are obsolete in general. Therefore, it may at first be necessary to determine the processor holding the valid page before propagating it to the requesting processor.

4.1 *Solutions in a FORCE-environment*

(a) *Broadcast solution.* The simplest treatment of buffer invalidation using a FORCE-propagation would be to broadcast the identifier of modified objects to all processors before the modifying transaction commits. Then, invalidated pages have to be discarded from the buffers to avoid access to obsolete objects. The FORCE-scheme ensures that the latest version of an object can be read from disk when the object does not reside in the local buffer. The broadcast solution has the advantage that it is nearly independent from the synchronization method in use but it can cause potential bandwidth problems. Assume for example a system of 8 processors, a throughput of 1000 tps with 50% update transactions and 4 modified objects per update transaction. With a message size of 100 bytes, there are 1.4 MB/sec necessary only for these invalidation messages. Furthermore, this message overhead increases as a square fucntion of the number of processors.

With the primary copy approach, buffer invalidations can be treated by the lock managers without any extra communication! This can be done by maintaining additional information in the block entries of the global lock tables and within the lock request and lock response messages as will be shown in the rest of this section. The techniques to be described are also applicable in a CLM-scheme.

(b) *Timestamp solution.* With this solution a timestamp is assigned to each page describing the most recent modification of any object in the page. For modified objects, the responsible lock managers keep the timestamps of the latest modification in the global lock table. When a lock request should be issued, it is firstly checked if the corresponding page resides in the local buffer. If so, the timestamp of this copy is sent to the authorized lock manager together with the lock request. This enables the lock manager to detect a conceivable buffer invalidation. In the example in Fig. 9, processor P2 owns the primary copy authority for block B. It is assumed that B was most recently changed at time t2 by a transaction T. With the release of the X-lock required for the modification, T also communicates the modification time t2 to the lock manager at P2. This timestamp is stored in the block entry for B in the GLT of P2. Assume now that a transaction at P1 wants to access page B and that the buffer of P1 holds a copy of B with timestamp t1 (t1 < t2). Since this timestamp is sent to P2 along with the lock request message, P2 can recognize the buffer invalidation by inspecting the block entry for B in the GLT. A detected invalidation provokes a lock response containing the demand to read the page from disk.

To reduce the overhead of storing the timestamps in the global lock tables, the identifiers of modified pages along with the set of timestamps can be broadcast in certain intervals to allow each processor to become aware of buffer invalidations. After successful broadcast the timestamp (or even the whole block entries) can be removed from the global lock table.

The method described has the advantage that no

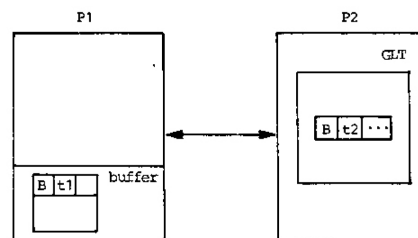


Fig. 9. Timestamp solution (example).

additional communication is required in the normal case. The drawback is the necessity to store a timestamp in every page. This can be avoided with the next solution.

(c) *Invalidation vector solution.* In this scheme a bit vector I (*invalidation vector*) is stored in the global lock table instead of using timestamps. Each invalidation vector I consists of N bits where N is the number of processors. The value of I(P) in the block entry of a block B indicates whether or not processor P may have an invalidated copy of B in its buffer. This information can be maintained because after a modification of block B at a processor P, only P has a valid copy of B in its buffer; for all remaining processors a buffer invalidation is possible. Therefore, I(P) is set to 0 when the X-lock on B is released; for all processors J ≠ P, I(J) is set to 1 because they may have an obsolete copy of B in their buffers.

With this scheme, each lock request from a processor P contains additional information whether or not the required page resides in P's buffer. If the local buffer holds a copy of the requested block, the PCA-lock manager can decide by using the invalidation vector whether this copy is up-to-date. The detection of a buffer invalidation is told within the lock response; in that case the block must be read from disk. When it is ensured that P gets the latest version of the block by reading from disk, the PCA-lock manager sets I(P) to 0.

In the above example for the timestamp solution assume now that transaction T which has modified page B was running on processor P2. Since P2 holds the PCA for B, the release of T's X-lock can be treated locally. When managing this lock release, the lock manager at P2 sets I = 10 in B's block entry of the GLT. This indicates that only processor P1 may have an invalidated copy of B in its buffer. When processor P1 issues a lock request for B in that situation and P1's buffer contains a copy of B, the lock manager of P2 can detect the buffer invalidation using the invalidation vector. Therefore, the lock response message also reports that the old copy of B has to be removed from the buffer and the new copy must be read from disk. P2 also sets I = 00 because P1 gets the latest version of B.

4.2 *Solutions in a NOFORCE-environment*

Although a FORCE-propagation simplifies the treatment of buffer invalidations it results in serious performance problems since forcing modified pages at EOT (in a synchronous manner) is overly expensive. Response times of update transactions are increased and the use of large database buffers is restricted since page modifications of different transactions cannot be accumulated before updating the database on disk[10]. Hence, for reaching high performance NOFORCE propagation should be the method of choice.

To cope with buffer invalidation using a NOFORCE-propagation scheme means to solve two subproblems:

(a) avoid access to obsolete objects in the local buffer and
(b) give the latest copy of an object to a requesting processor (transaction).

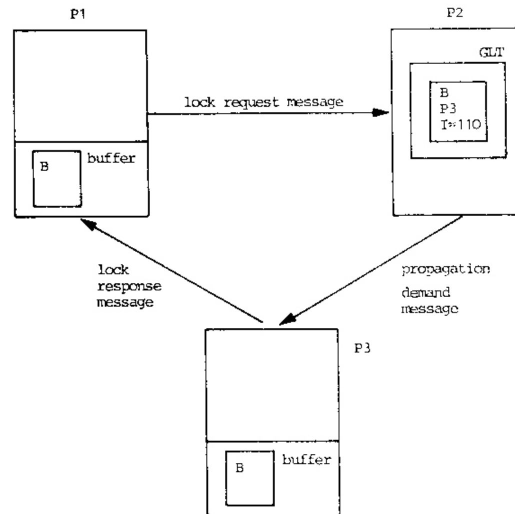


Fig. 10. Invalidation scenario with a NOFORCE-propagation.

For subproblem (a) all three solutions described above are applicable. However, we only consider the last solution which is clearly superior to the others. Subproblem (b) is new because with FORCE the latest copy could always be read from disk. With NOFORCE, however, the valid copy may be in any buffer or/and on disk.

A first solution with NOFORCE using invalidation vectors is to store also the name of the processor that has performed the latest modification of a block in the global lock table. For illustration assume that processor P2 owns the PCA for a block B, that was most recently changed by processor P3, and that processor P1 keeps an invalidated copy of B in its buffer (Fig. 10). In the block entry for B in the GLT of P2, P3 is kept as the processor having performed the last modification of B. The invalidation vector I indicates that P1 and P2 (but not P3) may have an obsolete copy of B in their buffer.

When now P1 issues a lock request message for block B to P2, it is told that there is a copy of B in P1's buffer. Using the invalidation vector, P2 recognizes that the copy at P1 is obsolete and that the correct version of B has to be propagated to P1 if the lock is grantable (a propagation would also be necessary if P1 had no copy of B in its buffer). Since P2 does not know whether the valid page is on disk or only in the buffer of P3, a message is sent to P3 (propagation demand) in order to arrange a correct propagation. P3 sends the lock response to P1 along with the valid page or with the demand to read the page from disk.

It should be clear that only with three different processors P1, P2 and P3 this procedure has to go through. A good partitioning and load balancing mechanism will achieve that most (read or update) accesses of pages are done at the processor with the PCA, resulting in simpler treatment and less communication delay for most lock requests. For example the simplest case would be if P1 = P2 = P3 for avoiding all messages; if P2 = P3 the propagation demand would not be necessary and if P1 = P2 the lock request message could be saved.

Another treatment of buffer invalidation with NOFORCE also uses an invalidation vector for modified pages in the global lock table and also requires information in the lock request messages whether or not the requested page is in the buffer of the sending processor. In difference to the former solution the valid version of a page is always found in the buffer of the PCA-processor or on disk. To guarantee this, update transactions must transfer each modified page to the PCA-processor during their EOT-processing (of course, communication is only required if the local processor does not hold the PCA). Although one needs no extra messages when the pages are piggy-backed to the release messages for the write locks, these messages are getting rather lengthy. The advantage of this strategy is that all lock requests for a page can now be managed by the PCA-processor without involving other processors.

This approach works best if there are no bandwidth problems and if the processors own large database buffers (for which conventional replacement strategies like LRU are applicable). Then, a fast exchange of modified pages (objects) is possible and a system-wide accumulation of updates can be performed without any physical writes. The efficiency (practicability) of this second proposal depends even more than the first solution on a good PCA distribution and on a coordinated load balancing scheme.

To get a feeling for the required bandwidths for the last solution, let us make a coarse estimation: Assume a throughput of 1000 tps with 50% update transactions. If each update transaction modifies 4 pages in average there are 2000 modified pages every second. If 80% of the updates are done by the PCA-processors and the page size is 2 KB then the exchange of modified pages requires 0.8 MB/sec. If only 50% of the updates could be done within the PCA-processors the bandwidth requirement would be 2 MB/sec. Of course, the message system still has to handle the synchronization messages requiring about 1 MB/sec (for 1000 tps, 10 locks per transaction, 50% local requests, 100 bytes for either lock request or lock response message). So, the total message traffic is clearly within the technically feasible range.

In the optimized version of PCL as described in Section 3, it is possible that lock managers not owning the PCA can locally grant S-locks. Since the PCA-lock manager is also responsible for detecting buffer invalidations, S-locks should only be granted by other lock managers if the respective page resides in the local buffer. Otherwise, when the block can't be found in the local buffer, communication with the PCA-processor must take place anyhow in order to get the valid page (the copy on disk may be obsolete!). So, 'empty' block entries in the LLT can be removed after the respective block is discarded from the local buffer.

5. CONCLUSIONS

In a loosely coupled DB-Sharing system the synchronization protocol must reduce the number of synchronization messages between the processors as far as possible. This can be achieved with the described primary copy approach if the centralized database is partitioned appropriately and if an effective load balancing can be performed. Besides of a basic version of the algorithm we have given some improvements in Section 3. A mechanism was introduced allowing effective handling of read locks thereby avoiding communication in most cases even for processors not owning the primary copy authority for the requested block.

In section 4, we presented a number of solutions to the problem of buffer invalidation. Most promising are the solutions using so-called invalidation vectors for avoiding any extra messages. The concept of the invalidation vector can be used in combination with a FORCE-propagation scheme as well as in a NOFORCE-scheme.

The optimized primary copy approach together with the invalidation vector solution to buffer invalidation has been implemented as part of a simulation system that should allow to quantify the performance behavior (throughput, response times) of the algorithm. The simulations are driven by real-life object reference strings[22] that represent different types of transaction load. The results of our simulations that are currently under way at the university of Kaiserslautern will be reported in a sequel to this paper.

Acknowledgements—I would like to thank T. Härder and P. Peinl for their helpful suggestions on an earlier version of this paper. The comments of the referees are also gratefully acknowledged. This work was financially supported by SIEMENS AG, Munich.

REFERENCES

[1] Anon *et al.* A measure of transaction processing power. *Datamation*, pp. 112–118 (April 1985).

[2] J. Gray *et al.* One thousand transactions per second. In *Proc. IEEE Spring CompCon*, San Francisco, pp. 96–101 (1985).

[3] W. Kim. Highly available systems for database applications. *ACM Comput. Surv.* **16**(1), 71–98 (1984).

[4] J. Gray. Why do computers stop and what can be done about it. In *Proc. Office Automation 85*, German Chapter of the ACM, pp. 128–145. Teubner (1985).

[5] T. Härder and E. Rahm. Classification of multiprocessor database systems—requirements, key concepts, implementation principles. Internal Report 152/85, Dept of Computer Science, Univ. of Kaiserslautern (in German) (1985).

[6] E. Rahm. Closely coupled architectures for a DB-sharing system. In *Proc. 9th NTG/GI Conf. on Computer Architecture and Operating Systems*, pp. 166–180, (in German) (1986).

[7] W. N. Keene. Data sharing overview. In *IMS/VS V1, DBRC and Data Sharing User's Guide*. Release 2, G30-5911-0 (1982).

[8] K. Shoens *et al.* The AMOEBA Project. In *Proc. IEEE Spring CompCon*, San Francisco, pp. 102–105 (1985).

[9] T. Härder. DB-sharing vs DB-distribution—die Frage nach dem Systemkonzept zukünftiger DB/DC-Systeme. In *Proc. 9th NTG/GI Conf. On Computer Architecture and Operating Systems*, NTG-Fachberichte 92, pp. 151–165 (in German) (1986).

[10] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* **15**(4), 287–317 (1983).

[11] A. Reuter. Load control and load balancing in a shared database management system. *Proc. 2nd Data Engineering Conf.*, pp. 188–197 (1986).

[12] E. Rahm. Algorithms for efficient load control in multiprocessor database systems. *Angewandte Informatik* **28**(4), 161–169 (in German) (1986).

[13] E. Rahm. Concurrency control in DB-sharing systems. In *Proc. 16th Annual GI Conf.*, Informatik-Fachberichte 126, pp. 617–632. Springer (1986).

[14] T. Härder and E. Rahm. Quantitative analysis of a synchronization algorithm for DB-sharing. In *Proc. of 3rd GI/NTG Conf. on Measurement, Modelling and Evaluation of Computer Systems*, Informatik-Fachberichte 110, pp. 186–201. Springer (in German) (1985).

[15] A. Reuter and K. Shoens. Synchronization in a data sharing environment. IBM San Jose Research Lab. (preliminary version) (1984).

[16] T. Härder, P. Peinl and A. Reuter. Optimistic concurrency control in a shared database environment. Dept of Computer Science, Univ. of Kaiserslautern/Stuttgart (1985).

[17] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.* **13**(2), 195–221 (1981).

[18] J. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science*, Vol. 60, pp. 393–481. Springer (1978).

[19] R. Obermarck. Distributed deadlock detection algorithm. *ACM TODS* 7(2), 187–208 (1982).

[20] A. K. Elmagarmid and M. T. Liu. Fault tolerant deadlock detection in distributed database systems. In *Proc. 15th FTCS*, pp. 240–245 (1985).

[21] E. Rahm. A reliable and efficient synchronization protocol for DB-sharing. Internal Report 139/85, Dept of Computer Science, Univ. of Kaiserslautern (1985).

[22] T. Härder, P. Peinl and A. Reuter. Performance analysis of synchronization and recovery schemes. *Database Engng* **8**(2), 50–57 (1985).