DESIGN OF OPTIMISTIC METHODS FOR CONCURRENCY CONTROL IN DATABASE SHARING SYSTEMS

Erhard Rahm

University of Kaiserslautern, FB Informatik Postfach 3049, D-6750 Kaiserslautern, West Germany

Abstract
Database Sharing refers to a local multiprocessor architecture where all processors share a common database at the disk level. Transaction systems running in such an environment intend to support high transaction rates with short response times as well as high availability and modular growth. In order to achieve these goals, Database Sharing (DB-sharing) requires an efficient synchronization component to control the processors' accesses to the shared database. In this paper, we concentrate ourselves on optimistic methods for concurrency control in DB-sharing environments that promise less synchronization messages per transaction than locking algorithms. We describe a new distributed protocol called broadcast validation where the validations for a transaction are simultaneously started at multiple processors by a broadcast message. Such a parallel validation scheme is prerequisite for reaching short response times and high transaction rates. The use of timestamps permits simple and fast validations as well as an integrated solution to the so-called buffer invalidation problem that is introduced by the DB-sharing architecture. Further improvements of our algorithm are proposed in order to reduce the synchronization overhead and to allow a combination with a distributed locking protocol which is advisable for applications with higher conflict probability. The communication and validation overhead of our algorithms is quantified by simple estimates.

1. Introduction

Future transaction processing systems for large applications, as in banking or reservation processing, will have to meet high performance and availability requirements. Such DB-based systems must be capable of high transaction rates (e.g. 1000 short transactions per second) with equivalent response times compared to present systems [11]. Another key requirement is extensibility of the system (modular growth).

A possible architecture for such high performance systems is Database Sharing (DB-sharing [14]) where a number of loosely or closely coupled processors have direct access to a shared set of databases. Each processor owns a local main memory and runs a separate copy of the operating system and the database management system. With loose coupling, interprocessor communication is exclusively based on messages, while closely coupled systems may use a common memory partition (e.g. as a global database buffer). In this paper, we focus on loosely coupled DB-sharing systems because we believe loose coupling offers the best framework to achieve high availability and extensibility. Examples of DB-sharing systems are the Data Sharing facility of IMS/VS [16], Computer Console's Power System [36], the DCS project [31] and the AMOEBA project [32].

Due to the physical attachment of the disk drives to all processors, the system components must be close together (e.g. in one room) permitting a high-speed communication system (1-100 MBytes/s). It is assumed that a global load control, performed by one or more front-ends (Fig. 1), distributes each incoming transaction to one of the processors (transaction routing). A transaction can be completely executed on one processor because each CPU has direct access to all parts of the shared database(s). This avoids the necessity of a distributed 2-phase-commit protocol, which would be required in a distributed database system where the database is partitioned among the processors. For transaction and crash recovery, a local log (not shown in Fig. 1) is maintained by each processor reflecting the updates of locally executed transactions. For media recovery a global log can be used that is constructed by merging the local logs [32].

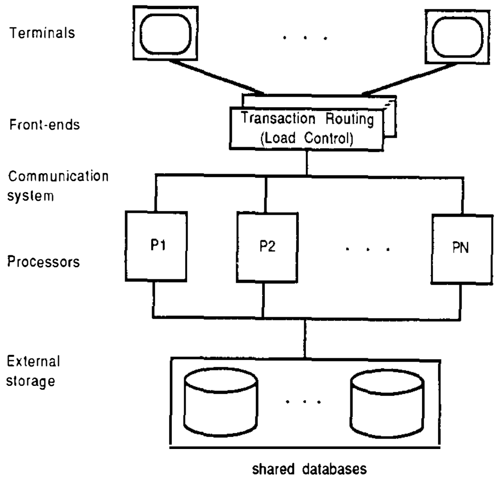

Fig. 1: Structure of a loosely coupled DB-sharing system

One of the main advantages of DB-sharing is flexibility. Since each processor can access the entire database, transaction load may be dynamically distributed according to current needs and system availability. Additional processors can be added without altering the transaction programs or the database schema. Likewise, a processor failure does not prevent the surviving processors from accessing the disks or the terminals. Transaction in progress on a failed processor are backed out and can be automatically redistributed among the available processors.

Obviously, DB-sharing needs a global **synchronization** protocol to control the processors' accesses to the shared database and to preserve serializibility of the executed transactions. Since there is no common memory, concurrency control requires message exchange among the processors that can seriously affect a transaction's response time; a high communication overhead does also limit transaction rates. Therefore, the concurrency control algorithm must minimize the number of synchronization messages per transaction. A survey of existing proposals for synchronization in DB-sharing systems can be found in [23,28].

Another difficulty with DB-sharing - the so-called **buffer invalidation problem** [26] - results from the existence of a local database buffer in each processor. Since a database page may simultaneously reside in multiple buffers, modification of any copy will thus invalidate all other copies. In order to solve this problem, invalidated objects in the buffers must be detected to avoid access to obsolete data. Furthermore, the latest object versions have to be propagated to other processors when they are requested there. The latter point is implicitly solved if a FORCE-strategy [15] is used for update propagation to the database on disk, i.e. all modifications of a transaction must be written to disk before the transaction commits. In that case, the most recent page version can always be read from disk. Unfortunately, the FORCE-strategy is not acceptable for high performance requirements due to the high I/O overhead that considerably increases the response times of update transactions. With a NOFORCE-scheme, however, it must be determined from where the latest page version can be obtained; modified pages may be exchanged across the shared disks or, preferably, via the interprocessor connections.

In this paper we investigate some new synchronization protocols for DB-sharing based on optimistic concurrency control that allow for an integrated solution to the buffer invalidation problem with NOFORCE. An optimistic concurrency control is particularly interesting for DB-sharing because no synchronization message is required during the processing of a transaction, but only for validation at the transaction's end. This should usually give better response times - provided the number of rollbacks can be kept small - than with a locking algorithm where multiple lock request messages per transaction are required, in general.
In the next section, we shortly review the original proposal of [17] for optimistic synchronization in centralized database systems and propose an improvement that uses timestamps for validation. After a discussion of known approaches for optimistic concurrency control in distributed environments in section 3, we describe our broadcast validation scheme that relies on distributed control. The algorithm also uses timestamps for conflict detection and performs all validations for a transaction in parallel. In section 5, we give a refinement of the broadcast validation scheme that should allow for a considerable reduction of validation and communication overhead, and in section 6 we show how the revised scheme can be combined with the primary copy locking algorithm, a distributed locking protocol for DB-sharing. Finally, we give a simple quantitative analysis of the synchronization overhead of our algorithms as well as some concluding remarks.

2. Basic Algorithms for Optimistic Concurrency Control

With optimistic concurrency control (OCC) a transaction consists of three phases: a read phase, a validation phase and a possible write phase [17]. During the read phase the updates of a transaction are performed within a private buffer not accessible by other transactions. Validation has to ensure that the execution of the validating transaction preserves serializability; conflict resolution relies on transaction abort as opposed to blocking in pessimistic (locking) algorithms. In the write phase, only required for successfully validated update transactions, sufficient log data must be forced to a safe place and modifications are made visible to other transactions.
Modifying database objects in private buffers allows for a simple undo recovery by just deleting the copies of uncommitted transactions (no I/O). Furthermore, concurrency is not reduced by blocking transactions, instead a consistent copy of an object is always accessible even if a modification for the object is in preparation. Another advantage is the deadlock freedom of OCC, thus saving an expensive deadlock detection usually required in locking algorithms.
In most OCC schemes the serialization order of transactions is determined by the validation order. Therefore, during validation it is checked that the validating transaction has seen all modifications of already validated transactions. In the original proposal of [17], a transaction number counter TNC is maintained to determine the transactions against which a transaction has to be validated. The TNC is incremented after each successful validation and its current value is used as the unique transaction number n(T) of the validating transaction T. For validation the system keeps track of the set of objects read and written by a transaction T (read set RS(T) and write set WS(T), respectively). In this paper it is assumed that the write set of a transaction is part of its read set. Let TSTART(Tj) be the value of the TNC at the start of transaction Tj, then the (serial) validation of [17] for Tj looks as follows:

```
VALID := true;
TFINISH (Tj) := TNC;
for n (Ti) = TSTART (Tj) + 1 to TFINISH (Tj) do;
    if RS (Tj) ∩ WS (Ti) ≠ { } then VALID := false;
end;
if VALID then do;    TNC := TNC + 1;
                     n (Tj) := TNC;
                     write phase for Tj;
end;
else abort Tj;
```

Validation and write phase form a critical section that prevents other transactions from validation; transactions in the read phase remain unaffected. An optimization allowing for parallel validation and write phases (however, at the expense of a higher restart probability) was also proposed in [17].

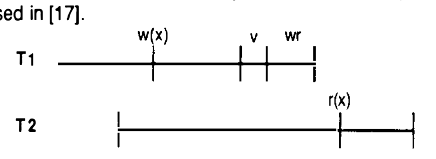

Fig. 2: Scenario with unnecessary rollback of T2

A disadvantage of the above validation scheme is that transactions are unnecessarily aborted in situations like in Fig. 2. In the shown scenario, validation of T2 fails because object x was modified by the concurrently executed transaction T1. However, the rollback of T2 is unnecessary since x was read after T1's write phase (wr) and thus the latest version was seen. The unnecessary rollbacks are due to the fact that the validation scheme of [17] does not regard the actual time interval that transactions run concurrently, but it assumes the worst case that they have completely been executed in parallel. This is especially unfavorable for long transactions that may often be (unnecessarily) restarted by short update transactions.
This problem can be circumvented by a revised scheme where timestamps are used to detect conflicts (the use of timestamps for validation was also proposed by other authors, e.g. in [34]). Here the transaction number of a successful update transaction T is stored as a timestamp or version number within each object modified by T. Now, a transaction has also to keep the timestamps of the accessed object versions in its read set that are used in the validation to decide whether an obsolete object version was seen or not. The validation of transaction Tj is as follows (TS (x) denotes the current version number of x, ts (x,Tj) is the timestamp of x as seen by Tj):

```
VALID := true;
for all r ∈ RS (Tj) do;
    if ts (r, Tj) < TS (r) then VALID := false;
end;
if VALID then do;
    TNC := TNC + 1;
    n (Tj) := TNC;
    for all w ∈ WS (Tj) do;   TS(w) := n (Tj);   end;
    write phase for Tj;
end;
else abort Tj;
```

Besides of the avoidance of unnecessary rollbacks, the revised scheme obtains two further improvements. First, it is no longer required to store the write sets of committed transactions, and second, validation is much faster because we now need only one comparision per read set element. A fast validation, however, is only given if we can access TS (x) in main memory which is not the case if object x has been replaced from the database buffer. This problem can be solved by reading TS (x) not from the object itself but from a separate main storage data structure (**'object table'**) where the timestamps of recently modified objects are kept. An entry for an object x can be removed from this table as soon as there are no more transactions that were running when the latest modification of x was performed. ** In a validation we now need only consider those objects with an entry in the object table.

Analytical studies [2, 20] as well as simulations [1,7,21] have revealed that only in nearly conflict-free environments the original proposal of [17] gives performance characteristics equivalent to locking; in most cases, however, the pessimistic approach performed better. The main reason for this lies in the risk of a high abortion rate or even cyclic restarts especially for long transactions or in the presence of (frequently modified) hot spot objects. Improvements are possible by the above described method based on timestamps or if a forward oriented synchronization scheme is used where validation is performed against running transactions and not against already committed ones [12,22]. However, these schemes do also not help in applications with a high conflict probability; here, the optimistic attitude is only feasible in combination with locking as proposed in [19,6,35]. Though more complex, these integrated strategies allow to combine the advantages of both approaches: a pessimistic synchronization can be used for long and already failed transactions to limit the number of rollbacks and to avoid cyclic restarts, while otherwise an optimistic concurrency control is applied to provide a high degree of concurrency and fast response times. First investigations have shown [11] that these integrated schemes have the potential to outperform pure locking algorithms.

The described validation scheme based on timestamps will be used in our distributed synchronization algorithms for DB-sharing systems that are developed by stepwise refinement in sections 4-6. Our final proposal will be a combination with a locking strategy to permit an efficient synchronization even for transaction loads with higher conflict probability. At first, however, we shortly look at some related work on OCC in distributed database systems and in DB-sharing systems.

3. Related Work on OCC in distributed environments

While there is a number of proposals for OCC in distributed database systems (e.g. [3,5,8,9,18,29,33]), only two optimistic methods were proposed for DB-sharing until now: a centralized scheme in [28] and a distributed algorithm based on a tokenring topology in [13,26]. Unfortunately, the known solutions for distributed database systems (that usually rely on distributed control) cannot directly be adapted to DB-sharing because of the buffer invalidation problem. Furthermore, in distributed database systems it is necessary to start subtransactions in order to access 'external' data and a distributed two phase commit protocol is required resulting in additional communication overhead compared to DB-Sharing where the read and write phases are performed locally. On the other hand, with a distributed validation scheme in DB-sharing systems a transaction has to validate at all processors in principle because there may be a conflict at any node, whereas in distributed database systems validation is restricted to the nodes where subtransactions were executed (i.e. a transaction can be synchronized locally if no external data was accessed). Compared to locking, the optimistic methods should usually cause less synchronization messages as a centralized locking algorithm that provokes communication for most or even all lock requests. With DB-sharing there are also less synchronous messages per transaction in general if a distributed optimistic protocol is applied instead of a distributed locking scheme (one validation request versus several external lock requests/releases). With NOFORCE, additional communication may be necessary in both approaches in order to fetch modified pages from external database buffers (see below).

A main shortcoming of a centralized validation scheme for DB-sharing is that the central node constitutes a single point of failure. Furthermore, a combination with a centralized locking scheme (necessary to limit the number of restarts) causes many lock request (release) messages for 'pessimistic' transactions. Bottleneck situations at the central validation site, however, can be eliminated - even for 1000 (short) transactions per second - by a sufficient fast processor (e.g. 30 MIPS) if timestamps are used for validation [23].

A much higher validation overhead is introduced by a distributed validation scheme where a transaction has to validate at each processor in principle; the total validation overhead increases therefore as a square function of the number of processors. In the proposal of [13] based on a tokenring topology, validations can only be performed at the processor possessing the token. Accordingly, after the read phase a transaction has to wait for the arrival of the token in order to validate locally. For validation against external transactions the read and write set of the transaction is transmitted around the ring along with the token, so that the transaction's outcome is not determined until the completion of a full ring circulation. The drawbacks of the tokenring approach are quite obvious:

- Response times are increased by the waiting time for the token arrival as well as by the time required for a further ring circulation. Additional processors do therefore increase response times.
- Throughput is limited because the validations are not performed in parallel but serially. Since the communication overhead grows with more processors we have less time to execute the drastically increased number of validations, thus restricting the possible numbers of processors and throughput.

Furthermore, the time t the token remains at a node must be carefully controlled, because short values of t (e.g. when only few transactions are ready to validate) cause a high communication overhead while higher values lead to longer circulation times and therefore to increased response times. Still worse, the whole algorithm collapses if t exceeds a certain value at one processor [13] because then all other processors produce even more transactions waiting for validation so that t constantly grows.

4. Broadcast Validation for DB-Sharing

In this section we describe a distributed version of the timestamp-based validation scheme of section 2 for synchronization in DB-sharing systems that avoids the disadvantages of the tokenring approach. The scheme is called 'broadcast validation' because the validations for a transaction are simultaneously started at all processors by broadcasting a validation request. The use of timestamps facilitates the treatment of buffer invalidations because access to obsolete objects can easily be detected. In the rest of this paper, we assume that synchronization takes place on block (page) level in order to allow an integrated solution to concurrency control and buffer invalidation. Furthermore, a NOFORCE-strategy is assumed for the database buffers; of course, the solutions can also be adapted to a FORCE-policy where the exchange of modified pages is simplified.

In contrast to the tokenring algorithm, with our broadcast validation scheme all local validations of a transaction are performed in parallel thus giving shorter response times and allowing for higher transaction rates. Furthermore, this approach is better suited to support modular growth because a transaction's response time is only weakly dependent on the number of processors. The algorithm uses a broadcast medium (e.g. a bus), that is feasible due to the local arrangement of the processors, to start the validations for a transaction simultaneously at all nodes. The results of the local validations are then returned to the processor P where the transaction was executed; if all local validations were successful the write phase for the transaction is performed on P, otherwise the transaction is aborted.

Since the local serializibility of a transaction at all nodes does not automatically guarantee global serializability [30] we enforce that transactions are validated at each processor in the same order. With a broadcast medium this requirement can be achieved quite easily by processing all validation requests in the order they were received, provided a reliable communication system is given. An alternative would be to assign a globally unique timestamp to a

** For this we keep for each transaction the current value of TNC at its BOT. The smallest value of these BOT values, that belongs to the oldest active transaction, is used to decide whether an object entry can be deleted from the object table.

transaction at the end of its read phase and to perform all validations in timestamp order; 'late' transactions (i.e. transactions with an EOT timestamp lower than that of already validated transactions) are aborted. Validating all transactions at all processors in the same order guarantees that a transaction successfully validated at each node does not affect global serializability; the global serializability order is determined by any of the local serializability orders which in turn are determined by the validation order.

In order to apply the timestamp-based validation scheme of section 2 it is necessary to assign each (successful) transaction a unique transaction number that indicates the transaction's position in the global serialization order. This can be done by maintaining a global counter (equivalent to the TNC) the current value of which is propagated and incremented with each validation request. Since each transaction is validated at each node and the validations are performed in the same order at every processor, a transaction can always get a unique transaction number that is greater than all previously assigned transaction numbers.

To perform the local validations an object table has to be maintained at each site; in this table an entry is created for each block successfully modified by a local transaction. The validation for a transaction T fails at processor P if there exists an entry in P's object table for a block B modified by a local transaction T' and if the transaction number of T' is greater than the object version of the copy of B accessed by T. In this case, T has seen an invalidated version of B. If the validation of T is successful at each node, then it is ensured that T has not accessed obsolete data.

Though the algorithm sketched so far detects if an invalidated object was accessed, it is preferable to prevent the use of obsolete pages as far as possible (i.e. to make the latest versions available) in order to minimize the amount of transaction aborts. To achieve this goal we can make use of the fact that the validation request for a transaction T contains T's write set that indicates the pages T is intending to modify. If T has been validated successfully at all nodes, all pages in the database buffers belonging to T's write set can be discarded because they become obsolete. After T's write phase the current version of these pages can be requested from the processor where T was executed (NOFORCE; with FORCE the pages can be read from disk). The information where the latest versions of modified blocks can be obtained is also kept in the object table.

Unfortunately, the fate of the validating transaction T is still uncertain after the successful validation at one processor and thus the write set of T indicates only possible modificatons. Since we should (optimistically) assume that T will not fail, it makes little sense to allow access to the pages belonging to T's write set, because the accessing transactions must then be aborted in case that T succeeds (T's modifications must be seen by all transactions validating after T). To avoid these unnecessary rollbacks we use a 'pessimistic' strategy and do not allow access to pages subject to a possible modification. For this purpose, the block entries of the object tables are extended to the following structure:

```
BLOCK-ID: ...
LAST-MODIFIER: transaction number of the latest successful
                   modifier;
MODIFYING-PROCESSOR: processor where the LAST-
                   MODIFIER was executed;
IN-DOUBT: Boolean;    (* indicates whether or not the block is
                   subject to a possible modification *)
POSSIBLE-UPDATER:  name of the transaction not yet com-
                   mitted that wants to modify the block;
WAITING-LIST: list of local transactions that wait until
               IN-DOUBT= false;
```

In order to access a block B at processor P during the read phase, a transaction T has now to apply the following procedure:

```
if (T's private buffer contains a copy of B) then access this copy;
else do;
    if (P's object table contains an entry for B) then do;
        if IN-DOUBT (B) then append T to the WAITING-LIST (B);
        else if MODIFYING-PROCESSOR (B) = P then read B
                from the local database buffer or from disk;
        else if (local database buffer holds a copy of B) and
                (timestamp of this copy = LAST-MODIFIER (B))
                then access this copy; (* B was already requested *)
        else request copy of B from MODIFYING-
                PROCESSOR;
    end;
    else read B from local database buffer or from disk;
end;
```

Note, that the 'locking' of pages by setting IN-DOUBT to true (corresponds to an X-lock) cannot result in deadlocks because the POSSIBLE-UPDATERs have finished their read phases and can therefore not be blocked by other transactions.

In order to activate blocked transactions, it is necessary to inform all processors about the final outcome of an update transaction, e.g by a broadcast message. These (broadcast) messages can be sent after the end of the update transaction and do therefore not increase the transaction's response time; these messages can also be bundled (e.g. with the next validation request) in order to reduce the communication overhead. However, there is a tradeoff between the reduction of communication overhead by bundling and the response times of blocked transactions, because the more the broadcast message about a transaction's fate is delayed, the longer are the waiting times for blocked transactions.

The informal description of the algorithm given so far is now specified more precisely by a procedural notation. The processing of a transaction T at processor P mainly encompasses the following steps:

```
read phase of T; (* T may be blocked sometimes due to
                    POSSIBLE-UPDATERs *)
broadcast validation request;
local validation of T with determination of n (T);
receive validation responses from other processors;
if (any validation failed) then abort T;
  else do;
    store n (T) as timestamp within the pages of WS (T);
    perform write phase for T;
  end;
if (T is an update transaction) then broadcast T's outcome to all
    processors;                     (* possibly delayed *)
```

If T was not successful, the failure of T need only to be communicated to the processors where T has been validated successfully. The (local) validation of T at a processor looks as follows:

```
VALID := true;
for all r ∈ RS (T) do;
  if (the local object table contains an entry for r) and
  ( ts (r,T) < LAST-MODIFIER (r) or IN-DOUBT (r) ) then VALID := false;
end;
if VALID then do;
   for all w ∈ WS (T) do;
        IN-DOUBT (w) := true; (* entry for w may have to be created
                                  at first *)
        POSSIBLE-UPDATER (w) := T;
   end;
end;
send validation response to the processor where T was
executed;
```

The algorithm shows that the validation of T also fails if IN-DOUBT holds for any of the referenced pages, because we assume that the POSSIBLE-UPDATER will be successful. A delay of T's validation until the actual fate of the POSSIBLE-UPDATER is known (to possibly avoid the rollback) is not advisable because then all further validations would also have to be delayed.

The broadcast information about the successful end of an update transaction does not only allow for the detection of obsolete pages in the database buffer and for the activation of waiting transactions, but it can also be used to save unnecessary work by aborting running transactions that are doomed to fail. These are all transactions that have accessed old copies of pages belonging to the write set of the successful transaction (this is possible in case the pages were accessed before they were locked). The early abortion of these transactions saves the work for the completion of their read phase as well as for their validation.

The following actions take place at a processor where T was successfully validated after the receipt of the message that indicates T's outcome:

```
if (T was successful) then do;
   for all w $\in$ WS (T) do;
      remove copy of w from the local database buffer if present;
                              (* copy obsolete *)
      IN-DOUBT (w) := false;
      LAST-MODIFIER (w) := n (T);
      MODIFYING-PROCESSOR (w) := processor where T was
                                                executed;
      abort all running transactions Tj with w $\in$ RS (Tj);
      activate the transactions waiting in WAITING-LIST (w);
   end;
end;
else do; (* T was aborted *)
   for all w $\in$ WS (T) do;
      IN-DOUBT (w) := false;
      activate transactions from WAITING-LIST (w);
   end;
end;
```

Of course, these actions are also executed at the processor where T was started. Here, however, the 'unlocking' of blocks (by resetting IN-DOUBT) and the activation of blocked transaction is delayed until the write phase of T is completed. The write phases can be performed in parallel because validation ensures that the write sets of concurrently writing transactions are disjoint (transactions cannot successfully validate if they have accessed blocks with IN-DOUBT = true).

In order to limit the number of entries in the object tables (and to avoid unsuccessful page requests), the processors keep track of locally modified pages that have been written out due to buffer replacement decisions. The information that the current version of these pages can now be read from disk is piggy-backed to the next broadcast message and is sent to all other processors. The corresponding block entries can simply be deleted from the object tables (note that these entries are not required for validation, because the transactions that have accessed obsolete copies of the pages are already aborted). This restricts the maximum number of object table entries roughly to the total number of buffer frames in the system. The entries should not be deleted from the object table except in the mentioned case, because otherwise information is lost where the current page version can be found. This, however, may lead to accesses to obsolete page copies and consequently to transaction aborts.

With the broadcast validation scheme described in this section, the only synchronization message that directly increases a transaction's response time is the validation request. Since all validations for a transaction are performed in parallel, the response time impact of the synchronization protocol should be quite small. Furthermore, the parallel validations should allow for higher transaction rates and better extensibility than with the tokenring approach. Further key properties of our algorithm were introduced to manage the buffer invalidation problem, to reduce the validation overhead and the number of rollbacks and to save unnecessary work:

- The use of timestamps permits the detection of access to invalidated page copies and a simple and fast validation. Furthermore, unnecessary rollbacks are avoided as shown in section 2.
- The blocking of pages subject to a possible modification help to limit the number of rollbacks and enables parallel write phases.
- Broadcasting the outcome of an update transaction that is required to activate waiting transactions allows to discard obsolete pages from the buffers and to store the information where modified pages can be obtained (thus reducing the number of aborts). Furthermore, running transactions that have accessed obsolete data can be instantaneously rolled back so that unnecessary work is saved. The broadcast messages are sent after the end of the update transactions and can be bundled to reduce the communication overhead.
- A single data structure, the object table, is used at each processor for validation and for blocking of pages, and to store the information from where modified pages should be requested.

5. Reduction of Validation and Communication Overhead

Though the broadcast validation scheme of the previous section allows for short response times, the communication and validation overhead is still rather high because a transaction is validated at each processor. As a consequence the synchronization overhead grows not linearly with the number of processors (N) but proportionally to N**2. Therefore, we propose a substantial refinement that should permit a noticeable reduction of the communication and validation overhead. The basic idea is to make a logical assignment of database partitions to processors and to validate a transaction only at those processors that are 'responsible' for at least one object of T's read set. This is the same principle used for OCC in distributed database systems where a transaction needs only to validate at the nodes where subtransactions were executed. With DB-sharing, however, the partitioning of the database is logical and only used for synchronization.

The same idea is applied in the **primary copy locking (PCL)** algorithm for DB-sharing [24] to reduce the number of external lock requests. For this purpose, the database is logically partitioned and each of the N processors is assigned the synchronization responsibility (= primary copy authority or PCA) for exactly one partition. With PCL, all lock requests for which one's own processor holds the PCA can be managed locally; only for the remaining objects a lock request message must be sent to the authorized processor. In order to keep the number of these messages low, the PCA distribution and the strategy of load control for routing transactions to the processors has to be suitably coordinated according to an assumed reference behavior of the transactions [27]. With such a scheme, transactions are usually assigned to that processor owning the PCA for most of the database portions the transaction is probably going to operate on (load control also has to ensure that no processor gets overloaded). Such a routing strategy does not only help to reduce the number of synchronization messages, but also to increase the hit ratio in the database buffers and to limit the number of buffer invalidations due to an improved locality of reference. The PCA distribution and/or the routing strategy can be dynamically adapted to changing conditions in the system (processor failure, new processor) or when the reference behavior of the transaction load has significantly changed.

In the rest of this section, we show how the broadcast validation scheme must be modified for a PCA-like synchronization in order to reduce the validation and communication overhead. In the next section we investigate a combination of the revised scheme with PCL to be preferably used in applications with higher conflict probability.

In the improved validation scheme the use of the object table is slightly different than in the previous section. Here, only the PCA-processor always knows exactly the LAST-MODIFIER for modified blocks, whereas the other processors are not informed about a modification until the broadcast message indicating the successful end of an update transaction is received. Consequently, the fields IN-DOUBT and POSSIBLE-UPDATER can only be maintained for blocks belonging to the local partition. For the identification of an object version, the name of the last modifying transaction is sufficient; i.e. transaction numbers are no longer required. This is because we already know that an invalidated page was accessed if the transaction name stored in the referenced page copy is different from the transaction name kept as the LAST-MODIFIER at the PCA-processor.

The validation of a transaction T at processor P looks now as follows (RS (T,P) and WS (T,P) denote the pages from RS (T) and WS (T), respectively, for which P holds the PCA):

```
VALID := true;
for all r ∈ RS (T,P) do;
    if (P's object table contains an entry for r)
        and ( ts (r,T) ≠ LAST-MODIFIER (r) or IN-DOUBT (r))
    then VALID := false;
end;
if VALID then do;
    if RS (T,P) = RS (T) then (* T must only validate at P *)
        for all w ∈ WS (T) do;
            LAST-MODIFIER (w) := T;
            MODIFYING-PROCESSOR := processor where T was
                                                  executed;
        end;
    else for all w ∈ WS (T,P) do;
            IN-DOUBT (w) := true;
            POSSIBLE-UPDATER := T;
    end;
end;
```

The algorithm shows that each element of T's read set is only checked at the PCA-processor, while with the basic broadcast scheme each object was inspected at each processor. So the total validation overhead per transaction is now exactly the same as the validation overhead at one processor in the basic scheme. In other words, the validation overhead could be decreased by a factor of N. If T was successful at P and if T has only referenced objects from P's partition, we can immediately update the fields LAST-MODIFIER and MODIFYING-PROCESSOR. Only if two or more processors are involved in T's validation, the outcome of T is uncertain after the successful validation at P and we therefore have to set IN-DOUBT to 'true'.

As in the basic scheme, the successful end of an update transaction is broadcast to all processors to remove invalidated pages from the buffers, to store the information where the current page versions are available and to abort running transactions that have accessed invalidated copies. Resetting of IN-DOUBT and activating of blocked transactions, however, may only be necessary at the (PCA-) processors where the transaction has validated.Similarly, the failure of an update transaction is only notified to the (PCA-)processors where the transaction has successfully validated.

Some aspects of the improved algorithm are now illustrated by the example in Fig. 3. Fig. 3a shows the situation where the most recent copy of block B resides in the buffers of processors P1 and P3. The object tables (OT) in each processor indicate that the last successful modification of B was performed by transaction T1 at P3. At P3 that owns the PCA for B the object table also shows that no transaction has notified a possible modification (IN-DOUBT=false). Assume now, that a transaction T2 at P1 wants to validate and that B belongs to T2's write set. Assume further that T2 has also referenced objects of the local partition and that it therefore has to validate at P1 as well as at P3. Fig. 3b shows the situation when T2 has successfully validated at P3. Since T2 may still fail at P1, in P3's object table block B is kept as IN-DOUBT with T2 as POSSIBLE-UPDATER. Note, that setting IN-DOUBT prevents only transactions at P3 from accessing B, while at P1 and P2 the old version of B may still be referenced.

In Fig. 3c the scenario is depicted when all processors have been informed that T2 was successful. In the object tables the fields LAST-MODIFIER and MODIFYING-PROCESSOR are changed to T2 and P1, respectively, and in P3 IN-DOUBT is reset to 'false'. At P1 the new copy of B was written from the private buffer of T2 into the local database buffer thereby overwriting the old copy of B; at P3 the copy of B is discarded for being obsolete. If T2 had failed to validate at P1 or P3 then the situation of Fig. 3a would have been reestablished.

With the revised scheme the validation overhead grows only linearly with the number of processors; the communication overhead is also much smaller because fewer processors have to process a validation request and to send a validation response. Especially for short transactions and with an effective transaction

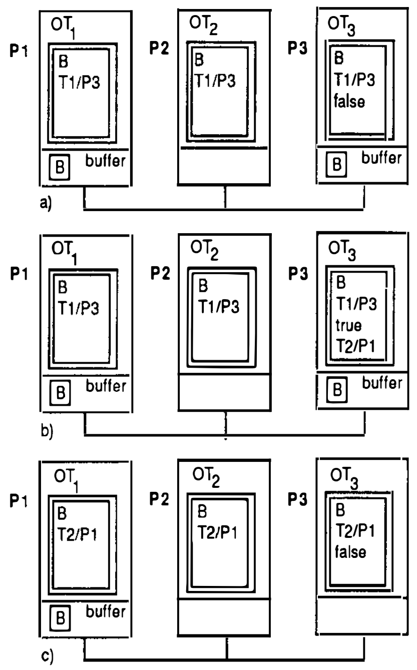

Fig. 3: Use of the object tables in the PCA-like OCC scheme (example)

routing strategy, many transactions may be completely processed and synchronized locally. The reduction of the synchronization overhead with the PCA-like synchronization (that is quantified in section 7) leads to a decreased competition for CPU-service at each node and therefore to better response times; the reduced overhead also facilitates modular growth and permits higher transaction rates. The number of restarts, however, may be a little higher than with the basic approach because access to pages that are possibly modified can only be prevented for pages of the local partition (by means of setting IN-DOUBT). Accesses to obsolete page versions are therefore somewhat more likely than before.

6. Combination with PCL

The improved broadcast validation scheme of the previous section should be sufficient for environments with low or medium conflict probability, e.g. if short or read transactions are dominating. However, if long update transactions and accesses to hot spot objects are frequent, there is the danger of a high abortion rate or even cyclic restarts. For these applications a combination of the broadcast validation algorithm with primary copy locking should be the best synchronization protocol where transactions are either optimistically or pessimistically synchronized. A pessimistic synchronization that guarantees the success of a transaction (except in case of deadlock) is advisable for already failed transactions (to prevent cyclic restarts) and for long (update) transactions for which a higher restart probability exists. Other transactions are optimistically synchronized to allow fast response times and a decreased communication overhead. The price for this flexibility, however, is an increased complexity of the protocol; in particular, global deadlock detection is more important because simpler deadlock resolution schemes (e.g. timeout) are expected to perform badly with higher conflict probability.

In the combined scheme, the block entries in the PCA-processor's object table have to be extended in order to keep information about granted and waiting lock requests. We assume that two lock types (read and write locks) are available and that locks are held until the end of the transaction. Pessimistic transactions prepare their updates also in private buffers in order to avoid access to 'dirty' modifications by an optimistic transaction.

The validation of an optimistically synchronized transaction T is mainly as in section 5 except that incompatible locks also lead to a validation failure (thus increasing the restart probability for optimistic transactions). So T's validation not only fails if an obsolete page version was seen or if IN-DOUBT holds for an element of RS (T), but also if a write lock is granted for any read set element or if a read lock is granted for any element of WS (T).

Lock requests are delayed due to an optimistic transaction only in case IN-DOUBT = 'true' because then the successful validation at the PCA-processor has been guaranteed. The processing of a lock request of transaction T for page x by the PCA-processor looks as follows:

```
GRANTED := true;
if   (entry for x exists) then do;
     if IN-DOUBT (x) or (incompatible locks granted) then do;
          GRANTED := false;
          append T to the waiting list for pessimistic transactions;
     end;
end;
if  GRANTED then do;
     adapt information about granted locks;
     send lock response if T is not a local transaction;
end;
```

Apparently, pessimistic transactions that are blocked due to IN-DOUBT have to be activated when the fatê of the POSSIBLE-UPDATER is known. In this case waiting optimistic transactions are only activated if there are no incompatible lock requests; otherwise, they are immediately aborted.
Since pessimistic transactions have to acquire a lock at the PCA-processor before they access a page, they can use the PCA-processor's information that is always up-to-date to get the latest copy of a block. In order to avoid unnecessary page requests, a pessimistic transactions checks before requesting a lock for a block B whether the local buffer holds a copy of B. If so, the version number of this copy is notified to the PCA-processor (together with the lock request) where it is decided whether this copy is up-to-date or not. If no copy was present or only an invalidated one, the current version of B can be requested from the MODIFYING-PROCESSOR when the lock is grantable.
When a transaction releases its locks, pessimistic transactions waiting for a lock can possibly be activated; for write locks the fields LAST-MODIFIER and MODIFYING-PROCESSOR are adapted in the object table. After the end of a pessimistic update transaction a broadcast message is also sent to all processors to specify modified pages. This information only relevant to optimistic transactions is used to abort running transactions that have accessed invalidated pages and to avoid further accesses to obsolete data.

7. Quantitative Assessment of the Validation and Communication Overhead

In order to quantify the superiority of the PCA-like synchronization scheme of section 5 over the basic broadcast validation algorithm, we give a simple estimation of the number of instructions required for communication (without requesting of pages from external buffers) and validation. For this purpose, we use the following parameters:

N	number of processors
T	transaction rate per processor (#transactions per second)
f	share of update transactions ($0 <= f <= 1$)
p	average number of partitions referenced by a transaction with PCA-like synchronization ($1 <= p <= N$)
L	average number of instructions for processing one transaction without synchronization
K	average number of instructions for sending or receiving one message
V	average number of instructions per validation (inclusive adaption of data structures)

For simplicity we assume that sending and receiving a message requires the same number (K) of instructions; we do also not distinguish between the send operation of a broadcast message and of a simple message.

The required number of instructions I per transaction (with synchronization) is as follows:

$$I = L + Isync$$

with $Isync = Icomm + Ival$.

Isync, Icomm and Ival stand for the number of instructions needed for synchronization, for communication and for validation, respectively. The (minimal) number of instructions per processor is $I \cdot T$; of course, the CPU capacity has to be sufficiently higher because CPU utilization should usually not exceed 80 %. The total number of instructions IT (ITsync, ITcomm, ITval) can be calculated from I (Isync, Icomm, Ivai) by multiplication with $T \cdot N$.
Let us first estimate the synchronization overhead for the basic validation scheme of section 4. Here, the validation request causes 1 send and N-1 receive operations; $2 \cdot (N-1)$ operations are required for sending and receiving of the validation results. Furthermore, the broadcast message to propagate the final outcome of an update transaction costs 1 send and N-1 receive operations. So we have

$$Icomm = [N + 2 \cdot (N-1) + f \cdot N] \cdot K = [(3+f) \cdot N - 2] \cdot K$$

For validation costs, we get

$$Ivai = N \cdot V$$

because a transaction is completely validated at each processor. For the PCA-like synchronization scheme, the validation overhead is merely

$$Ivai = V,$$

since each object is only checked at one processor. The communication overhead is also smaller because just p processors are involved in the validation of a transaction instead of N. Therefore, only (p-1) processors receive the validation request and send a validation response:

$$Icomm = [p + 2 \cdot (p-1) + f \cdot N] \cdot K = [3p + f \cdot N - 2] \cdot K$$

The differences are getting clearer, if we calculate the resulting overhead for 'typical' parameter settings:

$N = 2,5,10$
$T = 100$ (-> 200, 500 and 1000 transactions per second)
$f = 0.5$
$p = 1.2/ 1.8/ 2.5$ for $N = 2/5/10$
$L = 100000$
$K = 500$ or 5000
$V = 1000$

The value K=500 is typical for modern, message-oriented operating systems, whereas K=5000 is a realistic setting for large mainframe operating systems where the communication primitives are often quite expensive. The value of p usually depends on N (and the transaction load, of course), because with more processors it becomes increasingly difficult to support a high degree of locality within each processor. With the above settings we get the following instruction requirements (in MIPS) for synchronization:

	basic scheme			PCA-like synchronization		
	ITcomm		ITval	ITcomm		ITval
	K=500	K=5000		K=500	K=5000	
N=2	0.5	5.0	0.4	0.26	2.6	0.2
N=5	1.475	14.75	2.5	1.05	10.5	0.5
N=10	16.5	165.0	10.0	5.25	52.5	1.0

The table shows that the communication overhead dominates over the validation overhead for K=5000, so that in this case a bundling of messages is mandatory. The PCA-like synchronization allows for a drastic reduction of the synchronization overhead: for N=10 and K=5000 the synchronization overhead of the basic scheme is 165 MIPS (!), while the PCA-like synchronization requires less than one third (53.5 MIPS) of this overhead (note that only 100 MIPS ($N \cdot T \cdot L$) are needed for transaction processing without synchronization). For K=500 and N=10 the ratio is even 4:1 (26.5 MIPS versus 6.25 MIPS) confirming the clear superiority of the PCA-like synchronization.
In a similar way, the communication overhead for the primary copy locking scheme can be estimated. It turns out that usually the pessimistic synchronization does not only lead to more synchronous messages per transaction (thus giving increased response times), but that the total communication overhead is more

than twice as high than with the PCA-like optimistic synchronization scheme. In this comparision, however, the costs for deadlock detection and for requesting modified pages from other processors are not included. Furthermore, the actual communication overhead is, of course, strongly dependent on the transaction load.

8. Concluding remarks

In this paper we have proposed a distributed protocol called broadcast validation for optimistic synchronization in DB-sharing systems. The scheme was shown to be superior to known optimistic methods because all validations are performed in parallel thus permitting short response times; parallel validations are also prerequisite for high transaction rates and modular growth. The use of timestamps allow fast validations and an integrated solution to the buffer invalidation problem with a NOFORCE-strategy. The number of rollbacks could be restricted by blocking pages that will possibly be modified and by informing all processors about the fate of an update transaction in order to remove obsolete copies from the buffers and to make the modified page versions available. Furthermore, transactions having accessed obsolete data can be aborted early (during their read phases) thus saving unnecessary work.

In section 5, it was shown how a PCA-like synchronization can be applied to the broadcast validation scheme where a transaction only validates at the processors owning the PCA for at least one object of the transaction's read set. As demonstrated in section 7, the revised scheme drastically reduces the validation overhead (by a factor N) as well as the communication overhead. The revised scheme is not restricted to environments with a low conflict probability when it is combined with the primary copy locking algorithm as proposed in section 6. Such a scheme allows a transaction to be synchronized either pessimistically (in order to avoid a rollback) or optimistically (to reach short response times). The price for this flexibility, however, is an increased complexity of the protocol.

Additional improvements of our protocols are feasible if a multiversion scheme is used or if level-2-consistency is sufficient. As shown in [25], in these cases read transactions are always guaranteed to be successful because neither they have to validate nor must update transactions validate against them. This leads to a further reduction of the abortion rate and to improved response times for read transactions. Another point that could not be treated in this paper are the implications of a processor crash, in particular to study which provisions are necessary to properly continue concurrency control after a processor failure. These investigations are subject to ongoing research.

References

[1] Agrawal, R., Carey, M.J., Liny, M.: Models for Studying Concurrency Control Performance: Alternatives and Implications. Proc. ACM SIGMOD 1985, 108-121

[2] Agrawal, R., DeWitt, D.J.: Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation. ACM TODS 10 (4), 1985, 529-564

[3] Badal, D.Z., McElyea, W.: A Robust Adaptive Concurrency Control for Distributed Databases. Proc. IEEE INFOCOM 84, 382-391

[4] Bhargava, B.: Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems. Proc. 2nd Symp. on Reliability in Distr. Software and Database Systems, 1982, 19-32

[5] Boksenbaum, C., Cart, M., Ferrie, J., Pons, J.-F.: Certification by Intervals of Timestamps in Distributed Database Systems. Proc. 10th Int. Conf. on VLDB, 1984, 377-387

[6] Boral, H., Gold, L.: Towards a Self-Adapting Centralized Concurrency Control Algorithm. Proc. SIGMOD 1984, 18-32

[7] Carey, M.J., Stonebraker, M.R.: The Performance of Concurrency Control Algorithms for Database Management Systems. Proc. 10th Int. Conf. on VLDB, 1984, 107-118

[8] Ceri, S., Owicki, S.: On the Use of Optimistic Methods for Concurrency Control in Distributed Databases. Proc. 6th Berkeley Workshop on Distr. Database Management and Computer Networks, 1982, 117-129

[9] Ghertal, F.F., Mamrat, S.: An Optimistic Concurrency Control Mechanism for an Object Based Distributed System. Proc. 5th Int. Conf. on Distributed Computing Systems, 1985, 236-245

[10] Gold, I., Shmueli, O., Hofri, M.: The Private Workspace Model Feasibility and Applications to 2PL Performance Improvements. Proc. 11th Int. Conf. on VLDB, 1985, 192-208

[11] Gray, J. et al.: One Thousand Transactions per Second. Proc. IEEE Spring CompCon, 1985, 96-101

[12] Härder, T.: Observations on Optimistic Concurrency Control. Information Systems 9 (2), 1984, 111-120

[13] Härder, T., Peinl, P., Reuter, A.: Optimistic Concurrency Control in a Shared Database Environment. Manuscript, FB Informatik, Univ. Kaiserslautern/Stuttgart, 1985

[14] Härder, T., Rahm, E.: Multiprocessor Database Systems for High Performance Transaction Systems. Informationstechnik 28 (4), 1986, 214-225 (in German)

[15] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Comp. Surveys 15 (4), 1983, 287-317

[16] Keene, W.N. Data Sharing Overview. In: IMS/VS V1, DBRC and Data Sharing User's Guide, Release 2, G30-5911-0, 1982

[17] Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM TODS 6 (2), 1981, 213-226

[18] Lai, M., Wilkinson, K.: Distributed Transaction Management in JASMIN. Proc. 10th Int. Conf. on VLDB, 1984, 466-470

[19] Lausen, G.: Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking. Proc. ACM Annual Conf., 1982, 64-68

[20] Menasce, D.A., Nakanishi, T.: Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems. Information Systems 7 (1), 1982, 13-27

[21] Peinl, P.: Synchronization in Centralized Database Systems - Algorithms, Implementation and Quantitative Assessment. Doctoral Dissertation, Univ. Kaiserslautern, FB Informatik, 1986 (in German)

[22] Prädel, U., Schlageter, G., Unland, R.: Redesign of Optimistic Methods: Improving Performance and Applicability. Proc. IEEE 2nd Int. Conf. on Data Engineering, 1986, 466-473

[23] Rahm, E.: Concurrency Control in Database Sharing Systems. Proc. 16th GI Annual Conf., Informatik Fachberichte 126, Springer 1986, 617-632

[24] Rahm, E.: Primary Copy Synchronization for DB-Sharing. Information Systems 11 (4), 1986, 275-286

[25] Rahm, E.: Optimistic Concurrency Control in Database Systems: A Survey. Internal report 166/87, Univ. Kaiserslautern, FB Informatik, 1987 (in German)

[26] Rahm, E.: Integrated Solutions to Concurrency Control and Buffer Invalidation in Database Sharing Systems. Proc. IEEE 2nd Int. Conf. on Computers and Applications, 1987

[27] Reuter, A,: Load Control and Load Balancing in a Shared Database Management System. Proc. IEEE 2nd Int. Conf. on Data Engineering, 1986, 188-197

[28] Reuter, A., Shoens, K.: Synchronization in a Data Sharing Environment. Technical report, IBM San Jose Research Lab., 1984

[29] Schlageter, G.: Optimistic Methods for Concurrency Control in Distributed Database Systems. Proc. 7th VLDB, 1981, 125-130

[30] Schlageter, G.: Problems of Optimistic Concurrency Control in Distributed Database Systems. SIGMOD Record 12 (3), 1982, 62-66

[31] Sekino, A. et al.: The DCS - A New Approach to Multisystem Data Sharing. Proc. National Comp. Conf., 1984, 59-68

[32] Shoens, K. et al.: The AMOEBA Project. Proc. IEEE Spring CompCon, 1985, 102-105

[33] Sinha, M.K., Nanadikar, P.D., Mehndiratta, S.L.: Timestamp Based Certification Schemes for Transactions in Distributed Database Systems. Proc. SIGMOD 1985, 402-411

[34] Thomasian, A., Ryu, I.K.: Analysis of Some Optimistic Concurrency Control Schemes Based on Certification. Proc. SIGMETRICS 1985, 192-203

[35] Vidyasankar, K., Raghavan, V.V.: Highly Flexible Integration of the Locking and the Optimistic Approaches of Concurrency Control. Proc. IEEE COMPSAC 1985, 489-494

[36] West, J.C., Isman, M.A., Hannaford, S.G.: PERPOS Fault-Tolerant Transaction Processing. Proc. 3rd Symp. on Reliability in Distr. Software and Database Systems, 1983, 189-194

This work was financially supported by Siemens AG, Munich.