Concurrency Control in DB-Sharing Systems

Erhard Rahm

Dept. of Computer Science, University of Kaiserslautern
P.O.-Box 3049, D-6750 Kaiserslautern

Abstract:

In a database sharing (DB-Sharing) system multiple loosely or closely coupled processors share access to a single set of databases. Such systems primarily aim at high availability and high performance demanded by large transaction processing systems. To achieve high transaction rates with short response times an efficient concurrency control is required for synchronizing accesses to the shared database. This paper gives an overview of conceivable concurrency control algorithms for DB-Sharing. We distinguish between locking and optimistic methods and between centralized and distributed solutions. Five synchronization protocols are described in some detail and compared with each other.

1. Introduction

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). In the near future, such systems must be capable of processing 1000 transactions per second (tps) of the DEBIT-CREDIT-type [4,1] with equivalent response times compared to present systems. Further major demands are high availability [12], expandability (modular growth) and managability of the system.

It has been clearly recognized that monolithic systems (uniprocessors or tightly coupled multiprocessors) cannot meet these requirements for performance and availability reasons. More appropriate, however, are two basic multiprocessor approaches called DB-Distribution and DB-Sharing [10]. These systems consist of a set of autonomous processors that are loosely or closely coupled. Each processor owns a local main memory and a separate copy of operating system (OS) and DBMS. With loose coupling inter-processor communication is exclusively based on messages, whereas in closely coupled systems certain functions may be implemented using a common memory partition [10,17]. The difference between DB-Distribution and DB-Sharing results from the assignment of the disk drives to the processors:

- 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 DBMS such as R*.

- In DB-Sharing systems each processor has direct access to the entire database. This implies physical contiguity of all processors (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 [11], Computer Console's Power System [25] and the AMOEBA project [23].

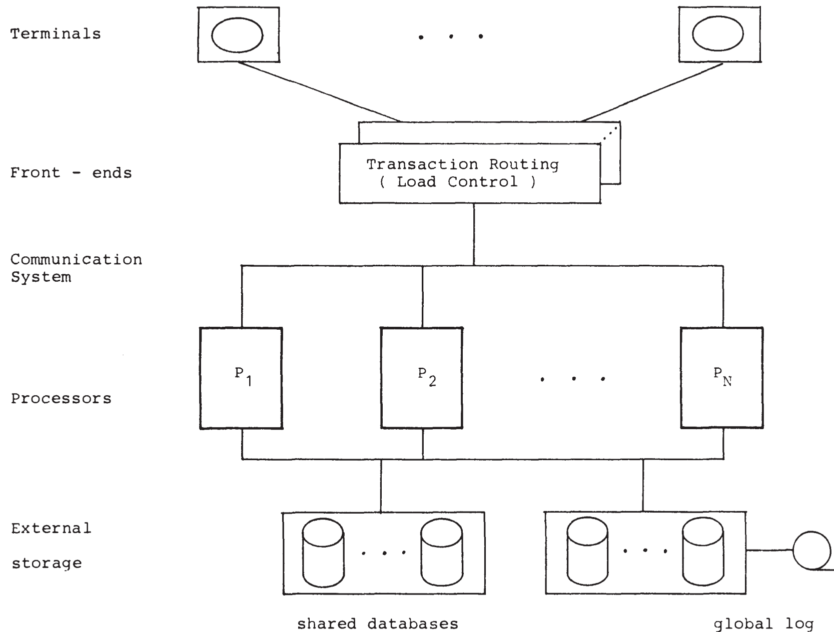


Figure 1: Structure of a loosely coupled DB-Sharing system

A detailed comparision between DB-Distribution and DB-Sharing can be found in [7] and [10]. Here, we concentrate ourselves on loosely coupled DB-Sharing systems as depicted in Fig. 1. A global load control, which may be located at one or more front-ends, distributes each incoming transaction to one of the processors (transaction routing). A transaction can be completely executed at 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 as required in DB-Distribution systems.

A main advantage of DB-Sharing systems 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 altering 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 on a failed processor can be rolled back and redistributed automatically among the available processors.

Naturally, some functional components must be redesigned compared to centralized systems in order to take advantage of the DB-Sharing architecture:

- The synchronization component has to coordinate accesses to the shared database thereby guaranteeing serializability of the executed transactions. To enable high transaction rates with short response times, concurrency control should require minimal communication between the participating processors. The realization of this component and the related problems are further discussed in the main part of this paper.

- Buffer control is needed to manage the problem of buffer invalidation that results from the existence of a local buffer in each processor. An update operation only modifies the processor's local copy of a database object; copies of the same object in other buffers are getting obsolete. Therefore, accesses to such invalidated objects must be avoided and a method to propagate the new contents of modified objects to other processors has to be supplied. For the latter point, there are two basic ways of exchanging modified objects between processors, namely via the inter-processor connections or across the shared disks.

  The simplest strategy to avoid access to invalidated objects within a buffer is to broadcast the identifier of modified objects to all processors before committing a transaction. So, obsolete copies can be detected and discarded from the buffers. The main deficiency of this general method is the large number of broadcast messages, especially in applications where update transactions are dominating. Much more efficient solutions are feasible if the synchronization component can cope with buffer invalidation, too [16].

- Load control has to find an effective strategy for transaction routing such that all processors are well utilized (however, without overloading any processor) and locality of reference (to decrease the amount of disk-I/O and buffer invalidation) is maximized. Furthermore, load control should reduce the number of global synchronization messages as far as possible. Obviously, the latter point requires a close cooperation between load control and concurrency control.

  Load control also has to react dynamically to changes in the workload and to the crash or reintegration of a processor.

- The recovery component is responsible for system-wide logging and recovery. Each processor has to maintain a local log (not shown in Fig. 1) required for transaction undo and crash recovery. Additionally, a global log (e.g. for media recovery) is constructed by merging the local log data. Crash recovery is performed by the surviving CPU's in order to continue transaction processing. Uncommitted transactions of the failed processor are backed out and restarted on another processor.

Concurrency control and load control are the most important functions in a DB-Sharing system for attaining high performance. In this work we present solutions to the synchronization problem, while load control is only discussed in the context of synchronization. In section 2 we give an overview of the concurrency control algorithms to be presented using a simple classification tree. It follows a description of the algorithms and a qualitative comparision of the different strategies.

2. Synchronization strategies for DB-Sharing

The fundamental problem of synchronization in a DB-Sharing system is due to the fact that concurrency control requires message exchange among the processors, because there is no common, instruction adressable memory which could hold control information for system-wide synchronization (e.g. a global lock table). These inter-processor communications are long and costly since sending and receiving messages are some of the most expensive operations in conventional OS. In order to reduce this communication overhead, messages may be buffered for transmission. This buffering, however, also increases waiting time for the response message. Additional overhead results from process or task switches for deactivating the calling process and activating the called process after receipt of a message.

Obviously, global synchronization requests are much more expensive than lock request handling in centralized DBMS (typically a few hundred instructions per request). Since transactions synchronously wait for the reply to a synchronization message, the above mentioned communication overhead and message buffering delays (transmission times are negligible on a high bandwidth network) increase response times directly. 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 transaction rates with acceptable response times are only reachable if the concurrency control algorithm minimizes the average number of synchronous messages per transaction. In general, this is only feasible in cooperation with load control (see below).

In this paper we present five different concurrency control algorithms for DB-Sharing that can be classified as shown in Fig. 2. We distinguish between locking algorithms and optimistic methods, both of which can be centralized or decentralized. Timestamp solutions are less attractive for DB-Sharing systems. Timestamps permit a fast decision about transaction ordering in case of contention, but our main problem is to detect access conflicts. Furthermore, the avoidance of deadlocks, guaranteed by timestamp protocols, is less important in a local environment compared to the price of a higher rate of transaction aborts as with a locking scheme.

A first survey of concurrency control in DB-Sharing systems can be found in the interesting paper [22]. In contrast to this work, we consider two additional schemes (Extended PTB, FOCC) and describe the BOCC-approach partially in more detail. Furthermore, we try a comparision of the algorithms with respect to general aspects like high performance, modular growth and availability.

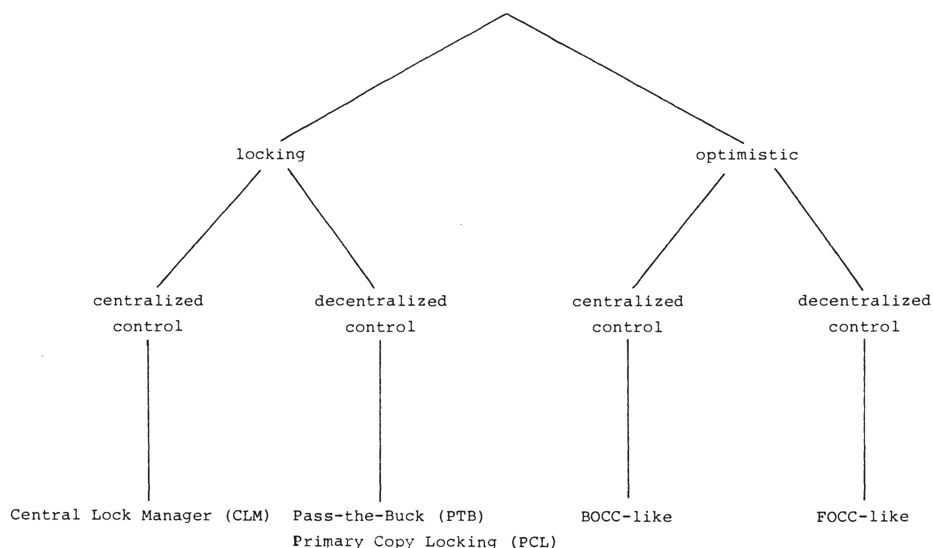


Figure 2: Concurrency control algorithms for DB-Sharing

The next section describes the three locking algorithms. Two of these schemes (CLM, PCL) are based on proposals for distributed DBMS [2,24]. In section 4 the optimistic methods are presented, and in section 5 we conclude with a comparision of the proposed algorithms.

3. Locking algorithms

Thus far, we have emphasized that a locking algorithm should minimize the number of global lock requests. For this, load control must route a transaction to that processor where most of the lock requests can be satisfied locally. Load control therefore needs a prediction of the presumable reference behavior of any incoming transaction. This is usually done by looking at the transaction type and possibly by analyzing input data [21].

In this section we do not discuss detection or avoidance of deadlocks because the techniques for that are the same as in distributed DBMS [2].

3.1 Central Lock Manager (CLM)

In this scheme the CLM resides at one processor and maintains a global lock table to answer lock requests from other CPU's. In the simplest form, each lock request is sent to the CLM 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 CLM shares the work with local lock managers located on 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 control as far as possible. In general, however, 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 [22].
The central lock manager approach is used in the DB-Sharing system of Computer Console. In their Power System 5/55, the CLM controls up to eight (medium-sized) application processors. A passive standby should overtake the synchronization responsibility in case the primary CLM fails [25].

3.2 Pass-the-Buck (PTB) algorithms

The basic form of the PTB-algorithm is used in the data sharing facility of IMS [11] which is restricted to DB-Sharing with two processors. In this approach, the two processors are alternately master of the system (decentralized control). Global lock information being kept by each processor may be altered only when the corresponding processor plays the role of the master. At the end of a master phase, a so-called buck is passed to the other processor. The reception of a buck indicates the beginning of a processor's master phase. In a buck there may be information of the following kind:

- modifications of the global lock information during the last master phase,
- lock requests which are not locally decidable and lock responses for such lock requests,
- notifications of modified objects to recognize buffer invalidation,
- other messages like commands of global interest.

In order to reduce the number of lock request messages, a so-called global hash table (GHT) is used. It contains a two-bit entry for each hash class of the lock table indicating which processor has interest in an object of the hash class. For instance, a 01-entry means that only transactions at processor 2 have interest. With the GHT a lock request at processor 1 can be locally satisfied if the respective entry has value 10, or if the value is 00 and processor 1 is master; in these cases, no conflict with processor 2 is possible.
An improved version of this approach called Extended PTB (EPTB) is presented in [9]. Though also designed for only two processors, it provides a number of enhancements:

- Use of a lock hierarchy
- Extended global lock information to reduce the number of lock request messages

- Effective treatment of buffer invalidation by introducing new lock types ('duration locks'). These new locks also try to exploit locality of reference to achieve a further reduction of global lock requests.
- The special case in which only one processor is permitted to modify the database (while the other can only issue read transactions) is particularly supported.

As simulations with a number of real-life page reference strings have shown, the algorithm provides good results especially with only one update processor [9]. In this case, throughput increase of factor 1.8 to 1.95 with response time degradations from 10 to 20 % compared to the single processor case could be obtained. In general, more than 95 % of the lock requests could be treated locally. As a consequence, short transactions issued less than one lock request message per transaction in average.

The scheme has the major drawback that many lock requests are satisfiable in the processor's master phase only. Therefore, response times are often increased by waiting for the next master phase in order to process a lock request. With more than two processors, these waiting delays would have been still worse.

3.3 Primary Copy Locking (PCL)

This approach 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. The database is therefore 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 [22]. As shown in Fig. 3, each lock manager maintains a global lock table (GLT) to control the objects of its partition. As opposed to the GLT, the local lock table (LLT) keeps information about granted or requested locks for local transactions only.

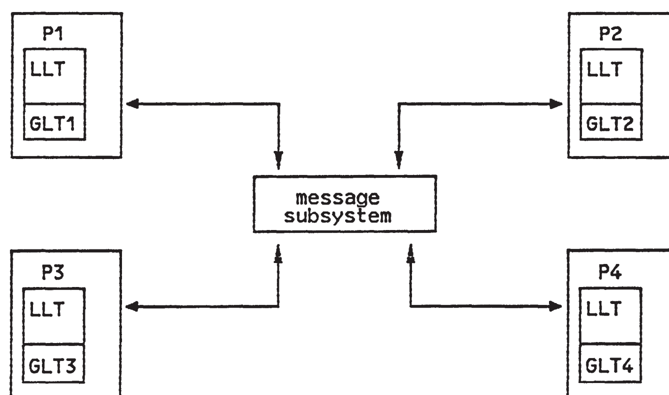


Figure 3: Primary Copy Locking (N = 4)

PCL 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 lock manager.

To take full advantage of the primary copy approach, transactions should not be routed to processors at random. Rather, load control has to 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 in case a processor fails or is added, or when the transaction load profile changes significantly [15]. Thus, PCL allows a tight and effective cooperation with the load control permitting flexible adaption to changing working conditions; these properties should result in much less messages for synchronization than using a CLM.

In [19] an optimization of the primary copy algorithm is proposed which provides a more effective treatment of read locks, especially for level-2-consistency (short read locks). Furthermore, solutions are given to cope with buffer invalidation using additional information in the GLT and avoiding any extra messages. Algorithms for a coordinated calculation of a routing strategy and a PCA distribution are presented in [18].

4. Optimistic concurrency control

With optimistic concurrency control (OCC) any transaction consists of a read phase, a validation phase, and a possible write phase [13]. During the read phase a transaction performs all updates within a private buffer not accessible by other transactions. The validation has to guarantee serializibility of the transactions; conflict resolution relies on transaction abort. The write phase is only required for update transactions which have successfully validated. In that phase, sufficient log data has to be forced to a safe place and the modifications are made visible to other transactions (update propagation).

In [6] two kinds of OCC schemes are distinguished: First, the backward-oriented approach (BOCC), originally introduced in [13], and second, the forward oriented method (FOCC).

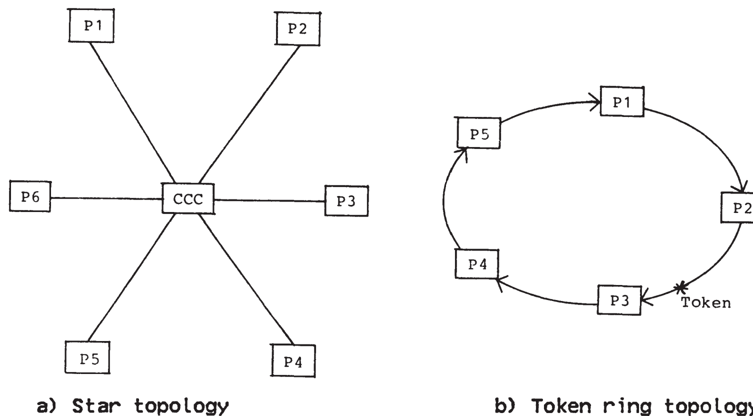
With <u>BOCC</u>, each transaction is validated against all committed transactions that have been running in parallel with the validating transaction at any point in time. Validation compares the read and write set of the validating transaction to the write sets of these completed transactions. In case of conflict, the validating transaction must be aborted.

With <u>FOCC</u>, on the other hand, the write set of a validating transaction is checked against the current read sets of all ongoing transactions. Accordingly, only update transactions have to validate. For conflict resolution, FOCC provides more flexibility than BOCC because all conflicting transactions are not yet committed [6].

In a DB-Sharing environment, OCC is particular attractive because long synchronization requests are necessary for validation only. Therefore, with BOCC the number of messages is restricted to one per transaction, and with FOCC to one per update transaction validation conflicts notwithstanding. Since the number of synchronization requests is fixed, OCC has somewhat different design goals than locking algorithms. A 'good' OCC scheme must mainly provide two characteristics:

1. Validation must be fast for attaining high transaction rates since only one transaction can be validated at any time.
2. The number of transaction aborts must be low since rolling back a transaction increases response time and induces additional overhead.

One problem with OCC in a distributed environment as DB-Sharing comes from the fact that validation has to be (at least logically) centralized to avoid that more than one validation is performed at any time. Furthermore, all updates of successfully validated transactions must be propagated in an uninterruptable way to ensure that other transactions see all the modifications or none. These requirements show the need of a centralized authority for validation and propagation control which can basically be implemented in two ways [8]:



a) Star topology b) Token ring topology

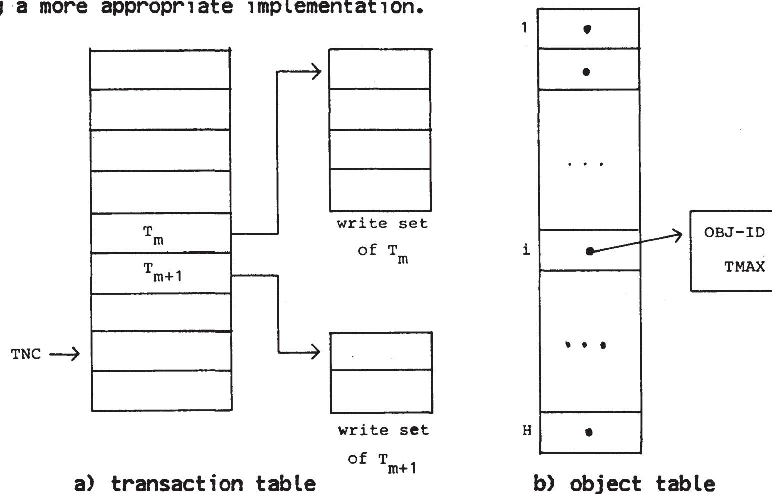
Figure 4: Synchronization topologies for OCC in distributed systems

- In a star topology (Fig. 4a), a central concurrency controller (CCC) can be used for all validations and also for controlling update propagation (and possibly buffer invalidation).
- In a token ring topology (Fig. 4b), on the other hand, control is distributed. Here, validations are only allowed at the site that owns the token which is circulating along a virtual ring connecting all processors.

In the remainder of this section we first describe a BOCC-scheme with a CCC. After that, a FOCC-like algorithm using a token ring topology is investigated.

4.1 Centralized BOCC-algorithm

In this scheme, each transaction has to send a validation request to the CCC after the read phase. The validation request contains the start time of the transaction as well as the read and write set. Before validation, the transaction is assigned a unique transaction number derived from a transaction number count (TNC). The TNC is increased by 1 each time a transaction number is assigned. Besides of the TNC, the CCC also maintains a so-called transaction table (Fig. 5a) to store the write sets of successfully validated transactions required for validation. The insertion order in the transaction table is determined by the transaction number.
To validate a transaction T, the CCC firstly uses the start time of T to find out the oldest update transaction T* that had run in parallel with T at any point in time. For validation of T each element of T's read set (we assume that the write set is part of the read set) must be compared to the write set of T* and to all write sets of transactions younger than T* being kept in the transaction table. Obviously, this simple implementation requires a huge number of comparisions per validation that grows with the transaction rate of the system. The total number of comparisions increases as a square function of the transaction rate. For this very reason, in [8] the whole approach is argued to be infeasible for high performance. It was estimated that the CCC would require 185 MIPS for validation of 1000 transactions per second (tps)! However, it is quite easy to eliminate the validation bottleneck by using a more appropriate implementation.



a) transaction table b) object table

Figure 5: Data structures of the CCC in a BOCC-scheme

Improved implementation

The number of comparisions can be drastically reduced by keeping an additional data structure named object table (Fig. 5b). The object table is organized as a hash table and contains an entry for each element of the write sets stored in the transaction table. Besides of the object identifier, a field TMAX is maintained in

each entry of the object table indicating the number of that transaction which has modified the object most recently.
The great advantage of the object table is that it allows for a fast validation, though additional storage requirement and maintenance overhead is introduced. With the object table it is no longer necessary to scan large parts of the transaction table in order to validate a transaction T. Now, for each element of T's read set one merely has to look up whether the object table contains an entry. A conflict is given if an entry is found with TMAX >= T* where T* is defined as above.

The usefulness of the improved implementation can already be demonstrated by a rough assessment of the processing capacity required for validation. The number of instructions per second (#vi) for validation can be estimated as follows:

$$\#vi = t * r * v$$

with
- t transaction rate (tps)
- r average size of a read set
- v number of instructions for looking up an entry in the object table and for comparing TMAX with T*

The formula shows that the number of instructions for a single validation does not depend on the transaction rate, but only on the size of the read set and on parameter v. The value of v can be kept small if the average number of entries per hash class is low, what can be attained by choosing the number of hash classes large enough. With t=1000 tps, r=10, and v=50, the above expression yields that the CCC must merely provide 0.5 MIPS for validation (instead of 185 MIPS) !

In a similar way, the number of instructions needed for maintenance of the data structures and for communication can be estimated. Although the processing capacity required for these actions is about one order of magnitude higher than for validation, the needed instruction rate also depends only linearly on the transaction rate. In our opinion, a 20-MIPS-processor (with a utilization of less than 50 % to keep contention small) should be sufficient for the work of the CCC in a DB-Sharing system with 1000 tps.

Further optimizations

In [22] two further improvements for the centralized BOCC-scheme are proposed in order to avoid multiple rollbacks of the same transaction and to deal with long transactions, for which a validation conflict is very likely:

- For transactions failed to validate, the CCC enters the read and write sets into the tables used for validation yet. This has the effect of preclaiming and guarantees that the second execution suceeds provided the same objects are referenced what is very likely for short transactions.
- It was further shown in [22] that the BOCC-scheme can be combined with a locking strategy where the CCC plays a similar role as the CLM discussed in 3.1. Although the whole protocol is getting more complicated, it also allows a 'pessimistic' synchronization advisable for long transactions in order to avoid cyclic

restarts. Since response time is not supposed to be critical for long transactions, the locking strategy seems to be appropriate for them.

4.2 Decentralized FOCC-algorithm

In this scheme, a processor can validate transactions in its master phase only (i.e. when the processor is hosting the token) guaranteeing that only one validation is possible at any time. Validation against local transactions can be done with the same techniques as in centralized systems. For validation against non-local transactions, the write sets of locally validated transactions (recall that only update transactions must validate) are sent along with the token (in a buck) to the other processors. Therefore, a processor must check all local transactions against these write sets in the buck after receipt of the token.
If a locally validated transaction T is permitted to be aborted at other processors, its write phase must be delayed until its write set has successfully passed all processors. This delay, however, can be avoided if the so-called kill policy is applied against external transactions. With this policy, all transactions are aborted that conflict with any write set in the buck. Therefore, the success of a transaction is guaranteed as soon as it has survived the local validation.
In such a scheme, the following actions take place after reception of a buck at processor P:

1. The write sets of transactions that have been executed on P are removed from the buck because they have just completed their round trip on the ring.

2. Global validation.
   The remaining write sets in the buck (originating from external transactions) are checked against the read sets of transactions currently in progress on P. If a conflict occurs, the local transaction is aborted (kill policy).

3. Local validation.
   Local update transactions having finished their read phases are validated against local transactions.

4. Update propagation.
   The updates of external transactions for which a global validation has been performed in step 2 must now be made visible at processor P. The write sets in the buck indicate the modified objects and can be used to detect invalidated copies in the local buffer [16].

5. Transmission of the token.
   The write sets of locally validated transactions are appended to the buck which is now forwarded to the next processor.

6. Write phases of locally validated transactions.

For transactions that have successfully validated in step 3 sufficient log data must be written and their updates are made visible to local transactions.

Steps 2 to 4 must run in a critical section.

To achieve high performance with this FOCC-scheme it is extremely important to keep the master phases as short as possible. In [8] it is even demonstrated that the whole algorithm is collapsing if the token remains longer than a certain time t at one node (the value of t is getting smaller with more nodes). The reason for this is that if the token remains too long at one node, all other nodes produce even more transactions waiting for global validation. Therefore, the validation periods after the receipt of the token are ever increasing what leads to the breakdown.

In order to avoid such problems, the write phases of local transactions (step 6) must be performed after the master phase (steps 1 to 5) because they require physical I/O (at least COMMIT-records of successful transactions must be written to the log). A further reduction of the master phases may be achieved if local validation (step 3) is partially performed before the token arrival. Although we cannot discuss possible realizations due to space limitations, it should be mentioned that this would considerably increase the complexity of the protocol.

Empirical studies for centralized DBMS have shown [14] that FOCC with a kill policy for conflict resolution leads to a high abort rate, especially for long transactions and in environments with a high share of update transactions. This should also hold for DB-Sharing systems where the global level of multiprogramming and hence the resulting conflict probability is even higher. Therefore, more flexible resolution strategies must be applied, in general, to avoid frequent restarts and to guarantee a fair scheduling. However, such alternative resolution strategies increase the response times of update transactions, because a validation now needs the agreement of all processors inducing an additional delay of one token round trip. Since the circulation time of the token raises with more nodes, only a few processors can be used in this approach due to the response time impact.

5. Conclusions

In the previous sections we have discussed 5 different concurrency control algorithms for DB-Sharing: three locking (CLM, EPTB, PCL) and two optimistic methods (BOCC, FOCC). Although sufficient quantitative analysis is still outstanding for most of the schemes, we will try to give a coarse comparision of the algorithms for a variety of aspects:

Response times

Response times are mainly determined by the average number of synchronization messages per transaction and - in particular with optimistic schemes - by the amount

of transaction abort. In algorithms based on a token ring topology (FOCC, EPTB), response time is also increased by waiting for the arrival of the token.
When concentrating on the number of messages, the optimistic protocols and the EPTB-algorithm are best, because they allow one message per transaction or less in average. The number of transaction aborts with BOCC and FOCC is not predictable; it primarily depends on the policy for conflict resolution, on the workload, and on transaction routing.
The CLM approach seems to require most synchronization messages since, in general, sole interest is not stable enough to provide a significant level of local synchronization. The PCL scheme is evidently superior to the CLM-design even so response times (and transaction rates) do heavily depend on the application and on the cooperation with the load control.

Transaction rates

The EPTB-approach can only support modest transaction rates (about 200 - 300 tps) for being limited to 2 processors. Among the other algorithms, the CLM approach appears to have the lowest performance due to the large message requirements. The performance of the remaining protocols is currently investigated in simulations.

Expandability and modular growth

Modular growth means that transaction throughput should grow almost linearly when adding a new processor whereas response times must not increase significantly. This is hard to achieve since the transactions at the new processor usually lead to a higher conflict probabilty and hence to more lock waits, deadlocks and transaction aborts.
The EPTB-protocol has been limited to two processors (no expandability) because otherwise waiting for the token arrival would have increased response times unacceptably. The token ring topology also restricts the FOCC-scheme to a small number of processors (less than 10) as explained in 4.2.
The CLM- and PCL-algorithms are also not likely to permit modular growth for more than a small number of processors. Due to the weakness of the sole interest concept, the CLM approach is restricted to few processors anyway; each additional processor makes sole interest situations even less probable. With PCL the distribution of the PCAs and the transaction routing must be adapted when a new processor is added in order to make use of the extended processing capacity. For modular growth, these adaptions - performed by the load control - must assure the same degree of local synchronization. This, however, is only possible if the number of significant transaction types is greater or equal to the number of processors and if these transaction types are mainly operating on distinct parts of the database. In our opinion, in many applications these prerequisites are only given for a small number of processors.
In the BOCC-scheme with the implementation technique described above, the CCC is not

likely to cause a bottleneck in the range of 1000 tps. The critical question with this scheme is whether or not the amount of transaction abort can be kept small enough when a new processor is added.

Dependence on load control

Load control has direct influence on system performance - irrespective of what synchronization technique is in use - since transaction routing determines the amount of buffer invalidations. For efficient synchronization, the locking algorithms (and mostly the PCL-scheme) depend more on the load control than the optimistic methods.

Availability

To provide high availability, concurrency control must guarantee the consistency of the database even in the presence of failures. Especially a processor crash must leave the database available for transaction processing at the surviving processors. Therefore, the concurrency control algorithm must be capable of correctly continuing synchronization after a processor failure.
Since the centralized algorithms (CLM, BOCC) offer single points of failure, provisions must be made for the crash of the central authority (CLM or CCC). The techniques for this are well-known from the design of fault-tolerant systems, e.g. one could install a shadow at a different processor that is periodically updated by checkpoint messages [5].
With distributed protocols, information essential for synchronization after a processor crash must also be maintained redundantly at different sites. For the PCL-scheme a method has been developed for reconstructing global lock tables that have been lost during a processor crash [15].

Hot spots and batch transactions

All proposed algorithms have problems to ensure high performance in the presence of hot spots and long (batch) transactions. In [4] it is advised to avoid hot spots in the database design, if possible. Batch transactions should be splitted into a number of mini-batches that can be run as a normal part of transaction processing.
If hot spots cannot be eliminated, one must use special protocols for synchronization. Such protocols based on locking are already proposed or implemented (IMS Fast Path) for centralized systems [3,20].

All in all, two of the five algorithms (CLM, EPTB) appear not to be appropriate to meet the requirements for a high performance DB-Sharing system. The other protocols are currently investigated in simulations driven by real-life database reference strings in order to quantify the performance behavior and to allow a more comprehensive comparision.

6. References

1. Anon. et al.: A Measure of Transaction Processing Power. Datamation, April 1985

2. Bernstein, P.A., Goodman, N.: Concurrency Control in Distributed Database Systems. ACM Comp. Surveys, 13 (2), 195 - 225, 1981

3. Gawlick, D.: Processing Hot Spots in High Performance Systems. Proc. IEEE Spring CompCon, San Francisco, 249 - 251, 1985

4. Gray, J. et al: One Thousand Transactions per Second. Proc. IEEE Spring CompCon, San Francisco, 96 - 101, 1985

5. Gray, J.: Why Do Computers Stop and What Can Be Done About It. in: Proc. Office Automation 85, German Chapter of the ACM, Teubner-Verlag, 128 - 145, 1985

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

7. Härder, T.: DB-Sharing vs. DB-Distribution - die Frage nach dem Systemkonzept zukünftiger DB/DC-Systeme. Proc. 9th NTG/GI conf. on Computer Architecture and Operating Systems, NTG-Fachberichte 92, VDE-Verlag, 151 - 165, 1986, in German

8. Härder, T., Peinl, P., Reuter, A.: Optimistic Concurrency Control in a Shared Database Environment. Manuscript, Univ. of Kaiserslautern/Stuttgart, 1985

9. Härder, T., Rahm, E.: Quantitative Analysis of a Synchronization Algorithm for DB-Sharing. Proc. 3rd GI/NTG conf. on Measurement, Modelling and Evaluation of Computer Systems, IFB 110, Springer-Verlag, 186 - 201, 1985, in German

10. Härder, T., Rahm, E.: Multiprocessor Database Systems for High Performance Transaction Systems. Informationstechnik 28 (4), 1986, in German

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

12. Kim, W.: Highly Available Systems for Database Applications. ACM Comp. Surveys 16 (1), 71 - 98, 1984

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

14. Peinl, P., Reuter, A.: Empirical Comparision of Database Concurrency Control Schemes. Proc. 9th Int. conf. on VLDB, 97 -108, 1983

15. Rahm, E.: A Reliable and Efficient Synchronization Protocol for DB-Sharing. Internal Report 139/85, Dept. of Comp. Science, Univ. of Kaiserslautern, 1985

16. Rahm, E.: Buffer Invalidation Problem in DB-Sharing Systems. Internal Report 154/86, Dept. of Computer Science, Univ. of Kaiserslautern, 1986

17. Rahm, E.: Closely Coupled Architectures for a DB-Sharing System. Proc. 9th NTG/GI conf. on Computer Architecture and Operating Systems, NTG-Fachberichte 92, VDE-Verlag, 166 - 180, 1986, in German

18. Rahm, E.: Algorithms for Efficient Load Control in Multiprocessor Database Systems. Angewandte Informatik 4/86, 161 - 169, 1986, in German

19. Rahm, E.: Primary Copy Synchronization for DB-Sharing. To appear in: Information Systems 11 (4), 1986

20. Reuter, A.: Concurrency on High-Traffic Data Elements. Proc. Principles of Database Systems, 83 - 93, 1982

21. Reuter, A.: Load Control and Load Balancing in a Shared Database Management System. Proc. 2nd Data Engineering Conf., 1986

22. Reuter, A., Shoens, K.: Synchronization in a Data Sharing Environment. IBM San Jose Research Lab., preliminary version, 1984

23. Shoens, K. et al.: The AMOEBA Project. Proc. IEEE Spring CompCon, San Francisco, 102 - 105, 1985

24. Stonebraker, M.: Concurrency Control and Consistency of Multiple Copies in Distributed Ingres. IEEE Trans. on Software Eng., SE-5 (3), 188 - 194, 1979

25. West, J.C. et al.: PERPOS Fault-Tolerant Transaction Processing. Proc. 3rd Symp. on Reliability in Distributed Software and Database Systems, 189 - 194, 1983

This work was financially supported by SIEMENS AG, Munich.