Concurrency and Coherency Control in Database Sharing Systems

Erhard Rahm

University of Kaiserslautern, Dept. of Computer Science

Technical Report ZRI 3/91, Dec. 1991,
Revised: March 1993

Abstract:

Database sharing refers to a general architecture for distributed transaction and database processing. The nodes of a database sharing system are locally coupled via a high-speed interconnect and share the common database at the disk level ("shared disk"). We discuss system functions requiring new and coordinated solutions for database sharing. In particular, the most relevant alternatives for concurrency and coherency control are classified and surveyed. We consider the techniques used in existing database sharing systems as well as algorithms proposed in the literature. Furthermore, we summarize previous performance studies on database sharing. Related concurrency and coherency control schemes for workstation/server database systems, network file systems, and distributed shared memory systems are also discussed.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems; C.2.4 [Computer-Communications Networks]: Distributed Systems

General Terms: Algorithms, Design, Performance

Additional Key Words: Database sharing, shared disk, concurrency control, coherency control, locking, optimistic concurrency control, cache coherency, buffer invalidation

1. Introduction

In this paper, we survey the main problems and the available solutions for the database sharing ("shared disk") architecture for distributed transaction and database processing. Database sharing assumes a locally distributed and homogeneous system where all processors reside in close proximity and run an identical version of the database management software. We discuss the database sharing approach primarily with respect to supporting high performance, scalability and high availability.

In business data processing environments, database processing is traditionally performed by centralized database management systems (DBMS) running on large mainframe computers. The workload is dominated by a set of pre-planned functions that are implemented by application programs. These programs interact with the DBMS to retrieve and update records in the database. Typically, an application program consists of multiple database operations that are executed as a single transaction by the DBMS. According to the transaction concept [Gr78, Gr81, HR83, GR93], the DBMS guarantees that transactions are completely executed or not at all (atomicity), that modifications of successful (committed) transactions survive system and media failures, and that users see a consistent state of the database despite concurrent accesses by other users. Most transactions are comparatively simple and access only a few database records; the typical resource consumption of such transactions ranges between 50,000 and a few million machine instructions and 0-30 disk I/Os. Due to the provision of high-level interfaces, however, an increasing workload share of more complex queries, e.g., for decision support purposes, is expected.

Growing demands for high performance, modular system expandability, and high availability require distributed transaction and database processing facilities. Large on-line transaction processing (OLTP) applications such as airline reservations or banking generate workloads of several thousand transactions per second [Gr85, HG89]. Since the CPU requirements for such high-volume applications exceed the capacity of the fastest uniprocessors, multiple processors must be employed. While CPU speed is improving at a high pace, the performance requirements grow even faster in many cases thus amplifying the need for distributed transaction processing. Furthermore, centralized systems face performance problems processing complex database queries that access large volumes of data [Pi90]. To keep the response time of such queries sufficiently small, multiple processors and parallel processing strategies must be utilized. Supporting modular system expandability or scalability requires that performance linearly be improved by adding new processors to the system (horizontal growth). High availability is another critical requirement for many business database applications which can only be met by distributed architectures [Ki84, GS91].

There are several architectural approaches in order to utilize multiple processors for database processing. Three general architectures termed "shared memory" (or "shared everything"), database sharing ("shared disk") and database partitioning ("shared nothing") are generally considered as most appropriate to meet the requirements discussed above [St86, Bh88, Pi90, DG92]. ***"Shared memory"*** refers to the use of multiprocessors for database processing. In this case, we have a tightly coupled system where all processors share a common main memory as well as peripheral devices (terminals, disks). Typically, there is only one copy of the application programs and system software like the operating system or the DBMS which is accessible to all processors via the shared memory. The shared memory supports efficient cooperation and synchronization between processors. Furthermore, load balancing is comparatively easy to achieve by providing common job queues in shared memory. On the other hand, shared memory can become a performance bottleneck thereby limiting the scalability of the architecture [DG92]. Furthermore, there are significant availability problems since the shared memory reduces failure isolation between processors and there is only a single copy of system software [Tr83, Ki84].

These limitations can be overcome by database sharing and database partitioning systems which are based on a loose coupling of processors. In loosely coupled systems, each processor is autonomous, i.e., it runs a separate copy of the operating system, the DBMS and other software, and there is no shared memory [Ki84]. Typically, inter-processor communication takes place by means of message passing. Loose coupling can be used for interconnecting uniprocessors or multiprocessors. We use the term ***processing node*** (or node) to refer to either a uniprocessor or a multiprocessor as part of a loosely coupled system.

Database partitioning and database sharing differ in the way the external storage devices (usually disks) are allocated to the processing nodes. In ***database partitioning*** or "shared nothing" systems, external storage devices are partitioned among the nodes. The database stored on the partitioned devices may be partitioned, too, or replicated. Each node manages its local database partition. Transaction execution is distributed if data from multiple partitions needs to be accessed. In this case, a distributed commit protocol is required to guarantee the all-or-nothing property of the transaction [CP84]. Furthermore, the query optimizer has to support construction of distributed execution plans if database operations should not be confined to data from a single partition.

In ***database sharing*** or "shared disk" systems (also called data sharing or DB-sharing), each node can directly access all disks holding the common database. As a result, no distributed transaction execution is necessary as for database partitioning. Inter-node communication is necessary for concurrency control in order to synchronize the node's accesses to the shared database. Furthermore, coherency control is required since each node buffers database pages in main memory. These page copies remain cached beyond the end of the accessing transaction making the pages susceptible to invalidation by other nodes. Similar coherency problems exist in workstation/server DBMS, distributed shared memory systems and network file systems.

A major advantage of database partitioning is its applicability to locally and geographically distributed systems. Conversely, database sharing typically requires close proximity of processing nodes and storage devices due to the attachment of the disk drives to all nodes. Furthermore, interconnecting a large number of nodes is more expensive for database sharing since every disk needs to be connected to every node.

A major problem for database partitioning is finding a "good" fragmentation and allocation of the database. The database allocation largely influences performance (communication overhead, node utilization) because it determines where database operations have to be processed. However, database allocations tend to be static due to the high overhead for physically redistributing large amounts of data. This reduces the potential for load balancing, in particular with respect to short-term workload fluctuations. Variations in the number of nodes (node failure, addition of new node) require a reallocation of the database. Database sharing avoids these problems since there is no need to physically partition the database among nodes. In particular, a higher potential for load balancing is given since each node can process any database operation [Ra92]. Several papers provide further discussions of the relative strengths and weaknesses of database sharing and database partitioning systems [Tr83, CDY86, Sh86, St86, Pi90, DG92, Ra92].

There are several commercially available systems and research prototypes for both architectures demonstrating their suitability for satisfying the performance and availability requirements. Well-known database partitioning systems include Tandem's Encompass and NonStop SQL products [Bo81, Ta89] and several database machines, e.g., Teradata's DBC/1012 [Ne86] and the Bubba and Gamma prototypes [Bo90, De90]. Existing database sharing systems and prototypes are the IMS Data Sharing product [SUW82], the Power System 5/55 of Computer Console [We83, Ki84], the Data Sharing Control System of NEC [Se84], the Amoeba prototype [Tr83, Sh85], Fujitsu's Shared Resource Control Facility [AIM86], DEC's DBMS and Rdb/VMS products within a VaxCluster [KLS86, JR89, RSW89, Jo91], and Unisys's Gladiator system [Gu92]. Recently, Oracle has introduced a version of its database product (called "parallel server") that supports database sharing on different hardware platforms [Or90]. Ingres also supports database sharing in VaxCluster environments. Furthermore, the IBM operating system TPF (transaction processing facility) supports a "disk sharing" for transaction processing, but without providing full DBMS functionality [Sc87, TPF88]. The largest reservation systems are currently based on TPF. They support thousands of transactions per second and more than 100.000 terminals [Sc87].

In this paper we concentrate on the database sharing approach for distributed transaction processing. In particular, the most relevant techniques for concurrency and coherency control are classified and described. These functions are of prime importance as they determine the amount of inter-node communication. Furthermore, coherency control can have a large impact on I/O performance. We do not discuss parallel query processing strategies since so far little work has been done on this subject with respect to database sharing [Ra93b]. As a consequence, we assume that all database operations of a transaction are completely executed at the node to which the transaction request has been assigned.

Section 2 contains a more detailed description of the system model assumed in this paper. It also discusses the functional components requiring new solutions for database sharing. In Sections 3 and 4, we present our classification schemes and survey the major approaches for concurrency control and coherency control, respectively. In addition to basic policies, we outline several extensions to reduce the communication frequency and to limit data contention. Furthermore, the main results of previous performance studies for database sharing are presented. Related concurrency and coherency control problems in workstation/server DBMS and other areas are discussed in Section 5. In Section 6 we provide a brief summary of this investigation. Three appendices summarize the concurrency and coherency control schemes that are used in existing database sharing systems or have been studied in relevant papers or performance studies.

2. System model and functional components

Figure 1 shows the general structure of a database sharing system. It consists of N processing nodes sharing access to all database and log disks. Shared access to the log files is needed to support crash recovery by surviving nodes (see below). Transactions initiated from terminals or workstations arrive over a wide-area or local-area network. They may be assigned to the processing nodes by workload allocation software running in front-end processors. Each node runs an identical version of the DBMS. We generally assume that communication takes place by means of message passing over a high-speed interconnect. An alternative considered in [DIY89, DDY91, Ra91, Ra93a] is the use of shared semiconductor stores for communication or storage of global data structures. Such an approach is also referred to as a "close coupling" and can improve performance albeit similar problems as for shared memory in tightly coupled systems may be introduced.

There are two major alternatives to interconnect the processing nodes with all disks:

- The traditional (IBM) approach is the use of multi-ported disks and multi-ported disk controllers that are connected with the node's I/O processors (channels). In existing systems up to 16 nodes may be connected to all disks in this way. Conventional interconnection hardware based on copper wires limits the maximal distance between processing nodes and disk drives to about 400 feet; new fiber-optic connections support distances of several kilometers [Gr92].
- The alternative is a message-based I/O architecture where messages are exchanged between disk controllers and processing nodes to perform I/O operations and transfer pages. In this case there is no inherent limit with respect to the number of nodes. Geographic separation of disk drives and processing nodes is feasible, but at the expense of increased access delays. DEC's VaxClusters offer such a message-based I/O architecture [KLS86] for currently up to 96 processing nodes and disk servers. Hypercube architectures such as Ncube [TW91] support an even higher number of nodes. Since they are based on microprocessors they also offer high processing capacity at low hardware cost[1].

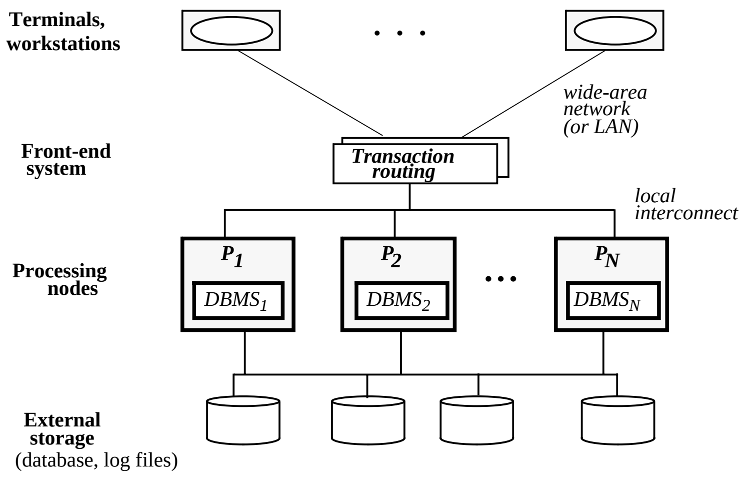


Figure 1. Structure of a database sharing system (simplified)

In the remainder of this section, we discuss the major functional components requiring new solutions for database sharing, namely concurrency control, coherency control, load control, and logging and recovery.

Concurrency control

Since any data item of the shared database can be accessed by any node, database sharing requires global concurrency control to correctly synchronize concurrent database accesses. The widely accepted correctness criterion for database concurrency control is serializability. Serializability requires that the actual (concurrent) execution of several transactions be equivalent to at least one serial execution of the same set of transactions [EGLT76, BHG87]. While concurrency control is basically a local function with database partitioning (each node synchronizes accesses against its partition), loosely coupled database sharing systems require explicit message exchange for system-wide synchronization.

The choice of the concurrency control scheme is of critical importance for the performance of a database sharing system since it largely determines the amount of internode communication for transaction processing. To support high performance, the number of concurrency control messages has to be as low as possible. Especially critical are so-called ***synchronous*** messages which entail transaction deactivation until a response message is received (e.g., global lock requests). These messages not only increase overhead due to process switches, but they also increase a transaction's response time and thus data contention (more lock conflicts or transaction aborts). Data contention can be a limiting factor for throughput particularly in distributed environments, where generally higher multiprogramming levels than in centralized DBMS, have to be dealt with. To keep data contention sufficiently low, it is also important to support fine-granularity (e.g., record-level) concurrency control and possibly tailored protocols for so-called "hot spot" objects that are frequently modified [Re82, Ga85, ON86]. Existing DBMS often sacrifize serializabilty to reduce lock contention by only supporting a reduced consistency level [GLPT76].
In order to limit the communication frequency it is also essential, as we will see, to treat concurrency and coherency control by an integrated protocol. A further requirement for a practical concurrency control protocol is robustness against failures in the system, in particular against node crashes.

Concurrency control algorithms for database sharing are described in Section 3.

Coherency control

DBMS maintain buffers in main memory to cache database pages [EH84]. With large database buffers, caching can substantially reduce the amount of expensive and slow disk accesses by utilizing locality of reference. Unfortunately, there is a ***buffer inval-***

1. In 1991, Oracle's database sharing system on a Ncube system achieved the highest transaction rate at the lowest cost per TPS (transaction per second) for the TPC-B benchmark [Gr91]. On a system with 64 nodes, more than 1000 TPS at about 2500 $/TPS were achieved [Or91].

idation problem in database sharing systems since a particular page (block) may simultaneously reside in the database buffers of different nodes (dynamic replication of database pages in main memory). Thus, modification of the page in any buffer invalidates copies of that page in other nodes as well as the page copy stored on disk. The basic task of coherency control is to ensure that transactions always see the most recent version of database objects despite buffer invalidations.

A disadvantage of buffer invalidations is that obsolete page copies cannot be reused thus reducing buffer hit ratios. The total number of buffer invalidations generally increases with both the buffer size and the number of nodes [Yu87]. In addition, performance is degraded by extra messages that may be required for coherency control. Even without buffer invalidations, the replicated storage of pages in multiple buffers reduces hit ratios compared to centralized DBMS or database partitioning systems with the same aggregate buffer size. On the other hand, data replication in main memory permits multiple nodes to concurrently read the same data thus improving the potential for load balancing. Furthermore, a buffer miss does not necessarily imply that the page must be read from disk. Rather, a page may be obtained much faster from another node holding a copy of the page in its buffer.

The buffer invalidation problem of database sharing systems is analogous (although at a different level of the storage hierarchy) to the cache coherency problem in tightly coupled multiprocessors [YYF85, St90] and to the replication control problem in distributed databases [BHG87, GA87]. Recently, coherence problems for pages cached in main memory buffers have also been studied in the context of network file systems [Ho88, NWO88], in so-called distributed shared memory systems [LH89, BHT90, SZ90, NL91], and in workstation/server DBMS architectures [WN90, CFLS91, WR91, FCL92]. The latter proposals often have a flavor that is similar to approaches that were already developed for database sharing. It is unfortunate that the database sharing proposals were not considered in the related areas, but that some basic schemes were "reinvented" with minor variations.

In Section 4 we survey coherency control schemes for database sharing systems. In Section 5 we briefly discuss concurrency and coherency control for the related areas.

Load control

The main task of load control is transaction routing, that is the assignment of incoming transaction requests to the nodes of the database sharing complex. This workload allocation should not statically be determined by a fixed allocation of terminals and/or application programs to nodes, but should be automatic and adaptive with respect to changing conditions in the system (e.g., overload situations, changed load profile, node crashes, etc.). Effective routing schemes not only aim at achieving load balancing (to limit resource (CPU) contention), but also at supporting an efficient transaction processing so that given response time and throughput requirements can be met. A general approach to achieve this goal is ***affinity-based transaction routing*** which uses information about the reference behavior of transaction types to assign transac-

tions with an affinity to the same database portions to the same node [Tr83, Re86, YCDI87, Ra92]. Such an approach is feasible for typical OLTP workloads since the reference characteristics of the major transaction types can be determined by DBMS-internal monitors and are expected to be relatively stable [Re86].
Affinity-based routing strives to achieve what we call ***node-specific locality of reference*** which requires that transactions running on different nodes should mainly access disjoint portions of the shared database. This is a promising approach since improved locality of reference supports better hit ratios and thus fewer disk I/Os. Similarly, node-specific locality helps to reduce the number of buffer invalidations and page transfers between nodes. Furthermore, locality of reference can be utilized by some concurrency control schemes to limit the number of synchronization messages (see Section 3).

The achievable degree of node-specific locality of reference is not only determined by the routing strategy, but also depends heavily on the workload characteristics and the number of nodes. Node-specific locality is hard to obtain if the references of a transaction type are spread over the entire database, if there are database portions that are referenced by most transactions, or in the case of dominant transaction types which cannot be processed on a single node without overloading it. Additionally, the more nodes are to be utilized, the less node-specific locality can generally be achieved unless new transaction types and/or database partitions are also added to the system.

A more detailed discussion of transaction routing and a framework for classifying different approaches can be found in [Ra92].

Logging and recovery

Each node of the database sharing system maintains a local log file where the modifications of locally executed transactions are logged. This information is used for transaction abort and crash recovery. For media recovery, a global log may be constructed by merging the local log data [Sh85]. Existing database sharing systems either use mirrored disks to handle disk failures, or provide a tool for merging the local log files off-line. The latter approach, however, does not allow for fast recovery from disk failures thus limiting availability. An on-line construction of the global log can support faster media recovery, although such an approach is difficult to implement. Fast disaster recovery can be achieved by maintaining a copy of the database at a remote data center. The copy can be kept up-to-date by spooling the merged log data to the remote system [BT90, KHGP91].

Apart from media and disaster recovery, crash recovery is the major recovery issue that requires new solutions for database sharing. Crash recovery has to be performed by the surviving nodes (that use the local log of the failed node) in order to provide high availability. The realization of this recovery form depends on many factors - including the underlying protocol for concurrency and coherency control - that are discussed in more detail in [Ra89]. In general, lost effects of transactions committed at the failed node have to be redone while modifications of in-progress, hence failed, transactions may have to be undone. Even for crash recovery a global log may be required if redo recovery is necessary for pages that were modified at multiple nodes without updating the permanent database on disk [Ra89, MN91]. Special recovery actions may be necessary to properly continue concurrency and coherency control, e.g., reconstruction of lost control information.

A more detailed treatment of recovery in database sharing systems is beyond the scope of this paper, but can be found in [Ra89, Lo90, MN91, MN92b].

3. Concurrency Control in Database Sharing systems

Although there are close dependencies between concurrency control and coherency control, it is helpful to separate the description of the major design alternatives from each other as far as possible. This is done to improve the clarity of the presentation as well as the understandability of the basic building blocks within the algorithms. Furthermore, the entire solution spectrum becomes clearer by looking at which alternatives for concurrency and coherency control can be combined with each other (see Section 4).

For concurrency control in database sharing systems, many proposals for distributed database systems could be adapted [BHG87, CP84, ÖV91]. However, differences in the transaction execution model and dependencies on coherency control make most proposals less attractive as they would result in a high amount of inter-node communication. We therefore concentrate on those approaches that were specifically proposed for database sharing or which are used in current database sharing implementations. The relevant schemes are based on either locking or optimistic concurrency control and operate either under central or distributed control (Figure 2). The locking schemes are described in Subsection 3.1, while the optimistic approaches are covered in 3.2. Since all existing database sharing systems use locking, these approaches will be treated in greater detail than the optimistic schemes. In addition, even for centralized systems there are several implementation problems of optimistic concurrency control for which efficient solutions still need to be developed [Hä84, Mo92].

In Subsection 3.3 we briefly summarize performance studies on database sharing concurrency control.

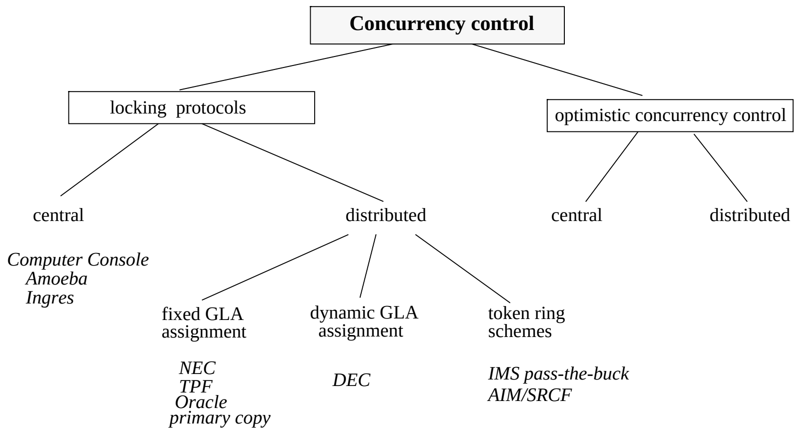


Figure 2. Concurrency control strategies for database sharing *(with example implementations)*

3.1 Locking protocols

In Subsection 3.1.1, we first describe several basic locking schemes for database sharing. Three of the protocols have in common that for every database object there is one lock manager in the system that is responsible for global concurrency control. There may be a single global lock manager for all database objects (central locking protocol) or the lock responsibility can be distributed among multiple lock managers on different nodes. In the distributed case, there may be a fixed or a dynamic assignment of the ***global lock authority (GLA)***[2]. A fourth approach is based on a logical token ring topology and requires a lock to be granted by multiple lock managers. In Subsection 3.1.2, three general enhancements are presented that may be able to reduce the communication overhead of the basic locking schemes.
The description assumes the use of read (shared) and write locks by transactions and that these locks are held until the end of a transaction (strict two-phase locking) [EGLT76]. We do not present approaches for global deadlock resolution because the same techniques as for distributed database systems can be used [BHG87, Kn87]. Due to close dependencies on coherency control, discussion of multi-version locking and record-level locking for database sharing will be deferred until Section 4.4.

3.1.1 Basic locking schemes

Central locking approach

In the central locking protocol, global concurrency control is performed by a single ***global lock manager (GLM)*** running on a designated node. The GLM maintains a global lock table to process lock requests and releases from all nodes. In the basic version of this approach, every lock request and release is forwarded to the GLM node. This requires 2 messages per lock request resulting in a very high communication overhead and response time increase. A single message at the end of a transaction is sufficient to release all locks of the respective transaction. Since the GLM knows the complete lock state of the system, central deadlock detection techniques can be used to resolve all local and global deadlocks.

It is comparatively easy to reduce the communication overhead by bundling multiple lock requests in one message. In this case, however, response times are not improved but likely to be higher than without such a message batching due to the delayed transmission of the synchronous lock requests. The enhancements discussed in 3.1.2 aim at reducing the number of global lock requests thereby improving both communication overhead and response times.

A general problem of central locking schemes is that the global lock manager is a single point of failure thus requiring special recovery provisions. Furthermore, the GLM may easily become a performance bottleneck thereby limiting the maximal achievable

2. In [LH89], a similar distinction between centralized, fixed distributed and dynamic distributed managers has been made for page ownership strategies in distributed shared memory systems.

transaction rates. To avoid that this throughput limit is very low and to keep queuing delays for lock processing small, it is generally advisable to reserve an entire processing node for global lock management.
Due to the simplicity of the central locking protocol, it could be implemented in hardware or microcode on a special-purpose processor. This promises a great reduction in communication overhead and delay provided special machine instructions are available to the processing nodes for requesting and releasing locks at the dedicated processor. The design of such a "lock engine" for database sharing is further discussed in [RS84, Ro85, IYD87]. In [Ra91, Ra93a], we examine the use of a fast shared intermediate memory for database sharing that could hold a global lock table accessible to all nodes. In such an architecture, a global lock request can be processed within a few microseconds so that a negligible delay would be introduced. Related papers on hardware support for (page-level) locking include [St84, CM988].

A central locking scheme is employed by several existing database sharing systems including the Amoeba prototype [Sh85], and the systems by Computer Console [WIH83] and Ingres. In Amoeba and Computer Console's system it is assumed that a standby process on a separate node can take over global concurrency control after a failure of the primary GLM. The new GLM can reconstruct the global lock table by collecting and merging lock information that is also maintained in the processing nodes. During the transition to the new GLM, no global concurrency control is performed, i.e., all database processing is "frozen" [WIH83].

Distributed locking with fixed lock authorities

A straight-forward *extension of the central locking scheme* is to distribute the lock responsibility to multiple global lock managers by partitioning the lock space, e.g., by using a hash function. This results in a distributed scheme with fixed lock authority. Assuming that each node knows the allocation of the lock authority (e.g., the hash function), lock request and release messages can be directed to the respective GLM with the same overhead as in the case of a single GLM. However, higher transaction rates should be possible since the lock/communication overhead is distributed among multiple nodes. On the other hand, message batching is probably less effective since there are now multiple destination nodes for global lock requests. Furthermore, multiple lock release messages are needed if a transaction has acquired its locks from different GLMs. In addition, information on waiting lock requests is dispersed across different nodes so that global deadlock detection becomes more complex.

Such an extension of the central locking approach is used in Oracle's and NEC's database sharing system and in TPF. The two latter systems rely on hardware support for global concurrency control. NEC uses a special-purpose node for global locking and deadlock detection. This lock engine internally consists of up to 8 processors that use a hash function to partition the lock space [Se84]. Each partition of the global lock table is kept in two copies so that after a processor failure one of the surviving lock processors can continue concurrency control on the respective database partition.

The TPF (transaction processing facility) operating system kernel supports disk sharing for up to eight processing nodes [Sc87, TPF88]. A rudimentary form of locking is performed by the shared disk controllers that maintain a lock table in their memory. The partitioning of the lock space is implicitly given by the data allocation to disks and the assignment of the disks to disk controllers. This approach provides "free" locking for objects that have to be read from disk since the lock request can then be combined with the disk I/O. For already cached data, however, a separate lock request (I/O command) must be sent to the disk controller[3]. The disk controllers only support exclusive locks on a per node basis rather than for individual transactions. Their lock table is of fixed size (512 entries) so that a lock request may be denied if the table is already full [BDS79].

Primary copy locking (PCL) is another distributed protocol with fixed lock authorities. The original primary copy scheme was devised for distributed databases with replication [St79]; its extension to database sharing is proposed in [RS84, Ra86]. As in the case of the extended centralized locking scheme, the primary copy protocol for database sharing requires that the lock space be divided into multiple partitions to distribute the lock responsibility among multiple nodes. However, PCL does not use dedicated nodes for global concurrency control but assumes that there is a GLM on each processing node (as part of the respective DBMS). Consequently, the number of lock partitions should correspond to the number of processing nodes. An advantage of this approach is that the system architecture is homogeneous since no special-purpose nodes are needed. Furthermore, lock requests against the local partition can be handled without communication overhead and delay. The GLA (global lock authority) allocation is known to all nodes[4].

A hash function could also be used for PCL to determine the GLA allocation such that each node controls about the same number of hash classes. If all hash classes are referenced with equal probability, we yield an average of (2 - 2/N) messages per lock request for N nodes. This is only slightly better than the central locking approach requiring 2 messages per lock request. The number of remote lock requests can be reduced for PCL by *coordinating GLA and workload allocations* such that transaction types are generally allocated to the node where most data references can be locally synchronized. If this is possible without overloading some of the nodes, many remote lock requests may be saved thus greatly improving performance. Furthermore, we yield a high degree of node-specific locality of reference where each node's partition is mainly referenced by local transactions. On the other hand, determination of suitable workload and GLA allocations is similarly difficult as finding an appropriate database allocation in database partitioning systems.

3. This is no problem for TPF since main memory caching of data beyond transaction commit is not supported. This also eliminates the need for coherency control. The disk controllers maintain a shared cache for all nodes, however.
4. In [RS84, Ra86], the term "primary copy authority" (PCA) has been used as a synonym for GLA.

Although we assumed so far that lock responsibilities are fixed, it is actually necessary to adapt the GLA in certain situations. This is particularly the case when the number of nodes participating in global locking changes, for instance, after a node failure. For PCL, it is also desirable to adapt the GLA allocation when significant changes in the load profile are observed in order to support (in coordination with workload allocation) a high share of local lock requests. Adaptation of lock authority allocations should be automatic to avoid the need of manual interactions by the database administrator. Such GLA adaptations are expected to occur comparatively seldom, since the number of nodes should vary infrequently and the load profile is typically stable for periods of several hours [Re86]. Consequently we have "almost" fixed allocations.

In contrast to the data allocation in database partitioning systems, the GLA allocation for PCL is only a logical data assignment represented by internal control structures. Hence, it can more easily be adapted than the physical data allocation for database partitioning. Furthermore, there is still a high potential for load balancing since the GLA allocation only determines the distribution of lock overhead, while the largest part of a transaction can be processed on any node. In database partitioning systems, on the other hand, the data allocation determines where the database operations, typically accounting for the largest part of a transaction’s path length, have to be processed.

Distributed locking with dynamic lock authorities

In the dynamic distributed locking schemes, the global lock authority for an object is not pre-allocated but dynamically assigned. Typically, the node where the first lock request for an object is requested obtains the GLA, i.e., its lock manager becomes the GLM for the respective object. The global lock responsibility may migrate between nodes depending on the ***migration policy*** that is implemented. Since each node knows for which objects it holds the GLA, no messages are required for requesting or releasing locks on these objects. However, for all other objects a ***locating method*** is needed to find out whether a GLM for an object already exists and on which node it is located. Therefore, in addition to the messages for requesting a lock from the GLM extra messages may be necessary to locate the object’s current GLM at first. These messages are avoided in the case of fixed lock authorities where all nodes know the responsible GLM for an object, e.g., by using of a common hash function or replicated data structure. It is also difficult to allocate the workload such that a high degree of local lock processing can be achieved since the assignment of global lock authorities is dynamic and therefore unstable (in contrast to the GLA allocations for PCL).

Migration and locating policies have already been investigated for the related problem of managing page ownerships in distributed shared memory systems [LH89, SZ90]. Here we concentrate on some variations that seem appropriate for database sharing. The GLA for an object should not migrate during the time an object is in use, i.e., between acquisition of the first and release of the last lock. This restriction guarantees that the node where a lock has been acquired is also responsible for releasing the

lock or processing a lock conversion (e.g., upgrade from a shared to an exclusive lock). Hence, no communication is needed for locating the GLM for lock conversions or unlock processing.
After the release of the last lock for an object, the GLA may also be released or it can be retained by the current GLM. Retention of the GLA seems preferable since it permits a local lock processing of object accesses that occur later in the GLM node. When a remote transaction issues the first lock request after a period of inactivity on an object, the GLA should migrate to the requesting node in order to permit a local lock processing there. When an object is not referenced for longer periods of time, the GLA can be released to limit the size of the lock tables.

For locating the current GLM for an object the use of a directory seems most appropriate. Since a central directory may introduce a performance bottleneck, the locating directory should be distributed among all nodes, e.g., by using a hash function to partition the lock space. Note that then up to 4 messages are necessary to acquire a lock: 2 to determine the GLM from the directory and 2 to request the lock itself from the responsible lock manager. In contrast, for the central locking schemes and PCL at most 2 messages per lock request are needed.
The messages for the directory lookup could be avoided by fully replicating the locating directory in all nodes. In contrast to the fixed allocation this approach seems not practicable here due to the larger size of the table and its high update frequency. So each time the GLA is assigned or released for an object, a broadcast message would be necessary to update all copies of the directory.

Dynamic GLA assignment is employed in the VMS operating system's Distributed Lock Manager (DLM) of DEC [KLS86, ST87]. The lock services of DLM are used by DEC's DBMS Rdb/VMS and VAX DBMS for database sharing in a VaxCluster environment [Jo91]. The locating method of DLM is based on a distributed directory that is partitioned among all nodes according to a hash function. To increase the duration of a GLA assignment, a hierarchical name space is supported where the global lock authorities are assigned for root-level objects (e.g., entire disks or record types). The GLM for a root-level object is responsible for lock processing on all sub-ordinate objects (pages, records, etc.) of the corresponding resource tree. As a result, the GLA is not released as long as there is a lock held on any object of the resource tree. What is more, a directory lookup is only required for lock requests on root-level objects while locks for subordinate objects can be requested directly from the GLM responsible for the corresponding resource tree. This can significantly reduce the probability of the worst case where 4 messages per lock request are needed. On the other hand, the use of coarse GLA allocation units can make it difficult to spread the lock and communication overhead equally across all nodes.

Token ring protocols

An alternative to the use of a single GLM per object is that all or at least a majority of the processing nodes grant a global lock request. A possible strategy for such an approach is based on a logical token ring topology where global lock requests are batched together with the token circulating through all nodes. Upon reception of the token, a node processes all global lock requests by checking whether or not there is a conflict with locally running transactions. The result of these checks is appended to the token so that a transaction knows about the outcome of a lock request after a complete ring circulation. The communication overhead of this approach is comparatively low since multiple lock request and lock response messages can be batched together with the token. Furthermore, the effectiveness of message bundling can be increased by holding the token in each node for a specific amount of time before forwarding it to the next node. On the other hand, the time until a global lock request is granted is very long and grows proportionally with the number of nodes. Hence, such a protocol may be useful for a small number of nodes only.

IMS Data Sharing uses such a token ring protocol called ***"pass-the-buck"*** for lock processing [SUW82, Yu87]. The protocol is restricted to two nodes. To reduce the number of remote lock requests a special hash table is used that is replicated in both nodes. For each hash class and node there is an "interest bit" in the hash table indicating whether or not transactions of a node have requested locks for objects belonging to a particular hash class. If the hash table indicates that only the local node has interest in a hash class, all lock requests for objects associated with the hash class can locally be synchronized. Modifications in the interest information are also propagated together with the token.

Fujitsu’s database sharing facility for its DBMS AIM uses a majority voting protocol where a lock must be granted by a majority of the nodes [AIM86]. A major disadvantage of such an approach is that no lock request can be granted locally.

3.1.2 Techniques to reduce the number of global lock requests

In the basic locking protocols described above, a global lock request could be avoided if the local node holds the GLA for the respective object. This idea is utilized in the PCL protocol as well as in the case of dynamic GLA assignments. Workload and GLA allocation should be coordinated to maximize the degree of local lock processing.

In this subsection, we present additional techniques to reduce the number of global lock requests. These approaches assume that there is a local lock manager (LLM) in each node (DBMS) that cooperates with the GLM for global concurrency control. An important extension of the basic locking schemes use special locks called read and write authorizations to authorize the LLMs to process certain lock requests and releases locally, i.e., without communicating with the GLM. A further extension is the use of a hierarchical locking scheme that aims at reducing the number of lock requests and messages by acquiring locks at a coarse object granularity if possible.

Write and read authorizations

Write and read authorizations are assigned by the GLM to LLMs and authorize them to grant and release locks locally:

- A ***write authorization*** is granted to a LLM when it requests a lock (on behalf of a local transaction) at the GLM and no other LLM (node) has issued a lock request for the respective object at this time ("sole interest" [RS84]). A write authorization for an object authorizes the LLM to locally process all read and write lock requests and releases by transactions running on the LLM's node. This is possible since the write authorization indicates that no other node has interest in the object so that no lock conflicts are possible with external transactions. Of course, a write authorization can be assigned to only one LLM at a time. It must be revoked by the GLM when a transaction of another node requests a lock for the respective object. These revocations are expensive since they introduce extra messages and delays for lock request from other nodes.
- A ***read authorization*** permits a local synchronization of read lock requests and releases and can be held by multiple LLMs at the same time. The GLM assigns a read authorization when a read lock is requested and no write lock has been requested at this point in time. A read authorization must be returned as soon as a write lock is requested for the respective object.

It is important to understand the conceptual difference between these authorizations and regular locks. While locks are requested and released by individual transactions, write and read authorizations are assigned to and revoked from entire nodes (LLMs). As a consequence, the authorizations are retained by a LLM beyond the end of the transaction whose lock request resulted in the assignment of the read or write authorization. This is done to permit a local synchronization for further lock requests that may be issued by local transactions in the future. The authorizations must be released by the LLM when the GLM revokes them because of a conflicting lock request. However in order to reduce the frequency of these expensive revocations, the LLMs should voluntarily release write or read authorizations when the respective objects have not locally been referenced for a longer period of time. This is also necessary to keep the lock tables sufficiently small to be kept in main memory.

The use of write and read authorizations is illustrated by the example in Figure 3. In the shown scenario, LLM2 on processing node P2 holds a write authorization on object O1 which is recorded in LLM2's local lock table as well as in the GLM's global

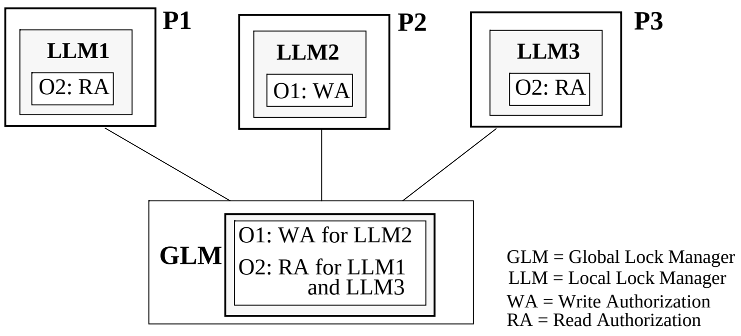


Figure 3. Lock request scenario with write and read authorizations

lock table. LLM1 and LLM3 are holding a read authorization for object O2. Therefore, all lock requests/releases for O1 by transactions in P2 and all read lock requests/releases for O2 in P1 and P3 can be handled by the local lock managers without communication. However, a lock request for O1 in node P1 requires 4 messages before the requesting transaction can continue processing. In addition to the two messages for the interaction with the GLM, two more messages are needed to revoke the write authorization from P2 thereby increasing the delay before the lock can be granted. A write lock request for O2 in P2 must be delayed until the read authorizations in P1 and P3 are released causing four extra messages.

The usefulness of write and read authorization depends on the degree of locality of reference between transactions (inter-transaction locality) of the same node. The authorizations pay off only if they allow more lock requests be locally satisfied than revocations occur. Write authorizations save global lock requests for objects that are accessed by multiple transactions of the same node that are not interfered by external transactions. Their effectiveness therefore requires node-specific locality of reference which is to be supported by an affinity-based transaction routing (Section 2). The effectiveness of read authorizations is independent of node-specific locality since they can be concurrently held in different nodes. However, their effective use depends on the amount of locality of read accesses which is determined by the workload. It is therefore advisable to use write and read authorizations selectively for certain object types. Read authorizations should be limited to objects with low update frequency. Write authorizations, on the other hand, are not appropriate for frequently referenced "hot spot" objects. For such objects it is unlikely that only one node has interest over longer periods of time so that frequent revocations would occur.

If both types of authorizations are to be supported, conversions between write and read authorizations can occur. For instance, if a write authorization is revoked because of a read request, it can be downgraded to a read authorization. For completeness, we summarize the major cases in Table 1. The upper table indicated the actions of the local lock manager upon a read or write lock request. The LLM's decisions depend on the local lock state for the object which may be NL (no lock requested so far), RA, WA or other (no local read or write authorization). The entry "GLM" indicates that a lock request message has to be sent to the global lock manager to acquire the lock. The GLM part of the protocol is summarized in the lower table. Here, lock state NL shows that the lock request is the first in the entire system for the respective object, and WA (RA) indicates that (at least) one node is holding a write (read) authorization. Lock state "other" means that no read or write authorization is granted indicating a situation where lock requests from different nodes are waiting for the lock. If a write (read) lock is granted to a transaction and there are already incompatible lock requests from other nodes known at the GLM at this time, no write (read) authorization is granted so that the lock is released at the GLM when the respective transaction commits.

	local lock state (at LLM)			
	NL	RA	WA	other
read lock request	GLM	grant	local (1)	GLM
write lock request	GLM	GLM	local (1)	GLM

	global lock state (at GLM)			
	NL	RA	WA	other
read lock request	grant RA	grant RA	revoke WA (2)	global conflict (wait)
write lock request	grant WA	revoke RA (3)	revoke WA (4)	global conflict (wait)

NL = no lock
WA = write authorization
RA = read authorization

(1) local lock conflicts possible
(2) assign RA (downgrade WA to RA), if possible
(3) assign WA (upgrade RA to WA), if possible
(4) assign WA, if possible

Table 1. Summary of global lock protocol with read and write authorizations

The use of write authorizations was first proposed in [RS84] (for a central lock protocol) under the name "sole interest". Read authorizations are due to [HR85, Ra86] where they have been proposed as an own concept which is independent of the use of write authorizations (sole interest concept). In [Ra86], the mechanism was called read optimization and described in detail for the primary copy locking scheme. In [HR85], it was shown that even a pass-the-buck protocol can utilize read and write authorizations to limit the number of global lock requests. Its interest bits mentioned above actually correspond already to write authorizations at the level of hash classes. In [Ha90], a distributed locking protocol for the Camelot transaction facility was outlined that supports data sharing between several "data servers". The scheme uses write and read authorizations and is based on a fixed GLA allocation (The authorizations are referred to as "cached locks" in write- or read-mode in [Ha90]. Revocation of an authorization is referred to as a "call-back".). Recently, Mohan and Narang have also described a central locking scheme for database sharing that uses read and write authorizations (termed LP locks) [MN92a].

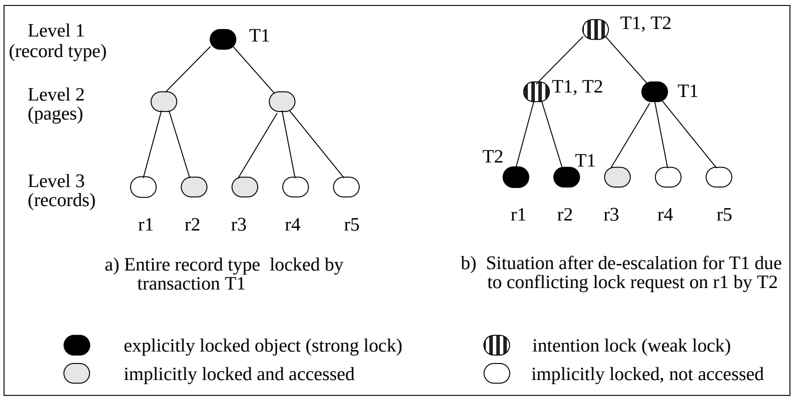
Hierarchical locks and authorizations

A well-known trade-off in centralized DBMS is the choice of the object granularity for concurrency control: a coarse granularity (e.g., files or record types) permits an efficient lock processing with little overhead but may cause high data contention (many lock conflicts and deadlocks) while fine-granularity locking (e.g., on pages or records) has the opposite properties. To support a compromise between the two contradicting goals, existing DBMS use hierarchical or multi-granularity locking schemes [GLPT76, BHG87, GR93] that support two or more object granularities.

For database sharing, choosing a coarser object granularity to decrease the number of lock requests (while permitting a sufficiently high concurrency) is even more important since it can also reduce the number of global lock requests to a large extent. Further message savings are feasible by using write and read authorizations at multiple levels of the object hierarchy. For instance if a node holds a write (read) authorization for an entire file, an implicit write (read) authorization is given for all pages of this file supporting a local synchronization on them. Since write and read authorizations can be used for all transactions of a node, they allow more message savings than a hierarchical locking protocol alone where a coarse-granularity lock only reduces the overhead for the owning transaction.

In [Jo91], a concurrency control scheme has been presented that dynamically adjusts the object granularity for lock requests according to the contention between running transactions. Assuming a hierarchical ordering of object granularities, the idea is to obtain a transaction lock at the coarsest granularity where it can be granted without lock conflict. If a transaction can acquire a lock at a coarse granularity (say a record type), all descendents in the object hierarchy (pages or records of the record type) are implicitly locked so that no global lock requests are required for them. If a lock conflict for a coarse-granularity lock occurs subsequently, the lock granularity for the lock holder is automatically reduced until there is no more lock conflict or until the leaf-level granularity (e.g., record) is reached. This process is referred to as ***lock de-escalation*** [Jo91]. Prerequisite for lock de-escalation is that after a transaction has obtained a coarse-granularity lock its locks at the finer object granularities are still recorded by the local lock manager to support a later refinement of the lock granularity. Hence, there is no improvement in terms of local lock maintenance overhead compared to a locking at the finest granularity, but global lock requests can be saved.

The main idea is illustrated by the example in Figure 4. In the first scenario, transaction T1 holds a "strong lock" (explicit read or write lock) on the record type thereby implicitly locking all pages and records of the record type. The local lock manager



Level 1 (record type)
Level 2 (pages)
Level 3 (records)
T1
r1 r2 r3 r4 r5
a) Entire record type locked by transaction T1
T1, T2
T1, T2
T1
T2
T1
r1 r2 r3 r4 r5
b) Situation after de-escalation for T1 due to conflicting lock request on r1 by T2
explicitly locked object (strong lock)
implicitly locked and accessed
intention lock (weak lock)
implicitly locked, not accessed

Figure 4. Hierarchical locking with lock de-escalation (example)

records that T1 has accessed records r2 and r3, although no explicit (global) lock request was necessary. When another transaction T2 wants to access record r1 in a conflicting mode, it begins requesting locks at the root level where a lock conflict with T1 occurs. This conflict starts the de-escalation process for T1 resulting in the situation shown in Figure 4b. The record lock for r1 has been granted to T2 since T1 has de-escalated the conflicting locks at the record type level and page level to "weak locks" or intention locks (these locks indicate that explicit locks are held at a lower level). The example illustrates that concurrency is not impaired compared to the case when all locks are requested at the finest granularity, although many global lock requests may be saved in the no-contention case.

The de-escalation technique can also be used for hierarchical write and read authorizations. For example in the scenario of Figure 3, if the conflict for the write authorization on object O1 is not at the finest granularity, the GLM would request a de-escalation for O1. In this case, LLM2 gives up the write authorization for O1 but retains write authorizations at the finer level for objects that have been accessed by transactions in P2 and that have not been requested by the conflicting transaction. Note that only global conflicts cause a de-escalation for authorizations. A strong coarse-granularity lock of a transaction, on the other hand, is already de-escalated after a local lock conflict so that this lock cannot be used any more to save global lock requests.

DEC’s database sharing systems employ the hierarchical locking approach based on lock de-escalation. However, only a limited form of read and write authorizations is supported by a feature referred to as "carry-over optimization" in [Jo91]. This technique is limited by the fact that the VMS Distributed Lock Manager keeps lock information for processes rather than for transactions and that there is a DBMS instance per process that does not share a global lock table with other DBMS instances of the same node. The DBMS does not release the read and write authorizations at the end of the transaction for which they were obtained but carries over these authorizations to subsequent transactions that are executed in the same process. Since there may be many processes in a node to execute transactions, the usefulness of read and write authorizations is significantly reduced compared to a node-wide maintenance of these authorizations. In particular, a local lock conflict can already cause a revocation (de-escalation) of a process-specific read or write authorization.

3.1.3 Combining the concepts

Table 2 shows which of the techniques to reduce the number of global lock requests can be employed in the basic locking schemes. A hierarchical locking scheme is generally applicable and recommended, and therefore not considered in Table 2.

The use of a local GLA is restricted to the primary copy locking protocol and the distributed locking schemes with a dynamic GLA allocation. Read and write authorizations can be employed by all locking protocols. The use of read authorizations is generally recommended since this optimization does not depend on node-specific locality so that it may improve scalability. In the distributed protocols based on a fixed or dy-

use of ... \ locking protocol	central locking	fixed GLA allocation (PCL)	dynamic GLA allocation	token ring (Pass the Buck)
local GLA	-	+	+	-
write authorizations	+	O	O	+
read authorizations	+	+	+	+

- not applicable O applicable / not recommended + applicable / recommended

Table 2. Applicability of techniques to reduce the number of global lock requests

namic GLA allocation, read authorizations permit a local read synchronization of objects being controlled by another node.
Write authorizations, on the other hand, appear not appropriate for the schemes that use a local GLA for reducing the number of global lock requests. This is because both concepts aim at utilizing node-specific locality of reference which should be supported by an appropriate transaction routing. However, GLA assignments are more stable than write authorization assignments since they do not change when multiple nodes need to access the same data. In order to support a local lock processing, it is therefore easier to allocate the workload according to the current GLA allocation than with respect to the assignment of write authorizations. In addition, the utilization of a local GLA for saving remote lock requests has no analogous disadvantage to the expensive revocations of write authorizations. Furthermore, assigning write authorizations would reduce the value of a local GLA since even at the GLM's node a lock request could not be granted until the write authorization is revoked.

Table 2 does not compare the different locking schemes with each other. The pass-the-buck scheme of IMS was one of the first protocols invented for database sharing, but it does not scale well. A central locking scheme introduces a potential bottleneck, which can easily be avoided by distributing the GLA among multiple nodes. Thus, the most promising locking schemes are based on either a fixed or dynamic GLA assignment combined with the optimizations discussed above. Dynamic GLA assignments are difficult to consider for affinity-based routing and may introduce extra messages to locate the GLM for an object. A fixed distributed GLA assignment can be used in combination with dedicated lock nodes or with locking perfomed on transaction processing nodes (PCL). The PCL scheme promises the greatest reduction of global lock request messages, but requires determination of suitable GLA assignments coordinated with affinity-based transaction routing.

3.2 Optimistic concurrency control

Optimistic concurrency control (OCC) methods have been proposed as an alternative to locking schemes, in particular for applications with low or moderate conflict potential [KR81]. In contrast to locking schemes, under OCC synchronization takes place at the end of a transaction where it is checked during a so-called validation phase whether there have been concurrency control conflicts with other transactions. Conflicts are generally resolved by aborting transactions. After successful validation, the modifications of a transaction (which are prepared on private object copies during transaction execution) are made visible for all transactions. There exist numerous variations of OCC in particular with respect to the validation schemes [Hä84, RT87].

For database sharing, the appeal of OCC is that only one global concurrency control request per transaction is needed, namely for system-wide validation. This promises a much lower communication overhead and delay than for locking schemes where multiple global lock requests per transaction may be required. Hence, the message frequency of the OCC schemes depends to a lesser degree on the workload profile, routing scheme and number of nodes. Furthermore, OCC schemes are deadlock-free avoiding the need for resolving global deadlocks. On the other hand, OCC schemes may suffer from a high number of transaction aborts considerably lowering performance (Throughput is reduced due to the waste of resources for executing transactions that fail during validation; response time is increased since a transaction may have to be executed several times before it can successfully be validated).
To limit the number of failed transaction executions it may be necessary to use a hybrid optimistic-locking scheme for concurrency control. For instance, if a transaction fails during validation it could use locking during the second execution to avoid repeated restarts. A different hyprid approach employs locking only for concurrency control within a node, while OCC is used to resolve global conflicts [RS84, Yu87]. Such a method promises a reduced communication frequency compared to a pure locking scheme and a lower number of transaction aborts compared to a purely optimistic approach. Affinity-based routing can reduce the number of conflicts between nodes and thus the frequency of validation failures [Yu87].
In the following we sketch different approaches for OCC in database sharing systems that are based on either a central or distributed validation approach.

3.2.1 Central validation

A straight-forward scheme uses a global validation manager (GVM) running on a designated node to perform all validations in the system. At the end of each transaction, a validation request is sent to the GVM which performs the validation and returns the validation result in a response message. Validation basically has to check whether the object versions that a transaction has accessed during its execution are still the most recent ones. If an object was accessed that has later been modified by a successfully validated transaction, validation fails and the transaction is aborted. The use of version numbers associated with database objects permits a very efficient implementation of such a validation [Ra87b].

To avoid repeated restarts of failed transactions, this scheme can nicely be combined with a locking approach [RS84]. The main idea is to request locks on all objects accessed during the first execution before restarting a transaction after a validation failure. These locks are requested at the GVM right after the failed validation so that no extra communication is needed. Furthermore, deadlocks can be avoided by requesting all locks in a pre-specified order. A lock set at the GVM prevents that an object can be modified by any other transaction. As a result, the re-execution of a transaction is guaranteed to be successful if no additional objects are accessed compared to the first execution. The second execution of a transaction is likely to be much faster than the first one since most database objects may still reside in main memory (reduced I/O frequency).
The idea of such a "lock preclaiming" for failed transactions can also be utilized for OCC in centralized DBMS [FRT90] and in database partitioning systems [TR90].

3.2.2 Distributed validation

A central validation component introduces availability problems and may become a performance bottleneck. Distributed OCC schemes try to avoid these problems by spreading the validation overhead among all systems. In [Ra87a, b], we discuss various alternatives for distributed validation that preserve global serializability. A simple approach uses a broadcast message to initiate validation of a transaction on all nodes in parallel. This scheme, however, creates a high communication and validation overhead that increases with the number of nodes. An improvement is possible when using an assignment of concurrency control responsibilities similar to the use of a GLA assignment for the primary copy locking scheme. In this case, a transaction needs only to validate at those nodes holding the responsibility for at least one of the objects referenced by the transaction. As outlined in [Ra87b], this scheme can also be combined with the primary copy locking protocol so that a transaction may be synchronized either pessimistically (using locking) or optimistically.

3.3 Performance studies

In this section, we provide a brief survey of the major performance studies that have already been conducted for database sharing. Although most studies considered both concurrency and coherency control, we concentrate here on concurrency control. Performance results with respect to coherency control will be surveyed in Section 4.5. Unfortunately, the results of different studies cannot directly be compared in most cases since there have been significant differences in the under-lying methodology, system models and workload characteristics. Appendix C provides an overview of 16 performance studies on database sharing. These studies are based on analytical modelling or simulations with synthetic or trace-driven workload models.

The performance of pass-the-buck protocols has been analysed in [HR85, YCDI87, Yu87]. These studies found that communication overhead and delay per global lock request are largely determined by the token holding time. Short token holding times result in high communication overhead and CPU contention, while longer token delays increase the waiting times for global lock requests and thus response times and lock contention. A "good" value of the token delay is therefore difficult to determine and should dynamically be adjusted according to the current workload situation (IMS uses a static value to be set by the database administrator). In [HR85] an extended pass-the-buck scheme has been investigated that uses hierarchical read and write authorizations at the level of files, hash classes and pages. In trace-driven simulations, these optimizations helped that generally more than 85% of all lock requests could be locally granted for two nodes if an affinity-based routing is employed. In [YCDI87], only a sole-interest concept for hash classes was considered (referred to as hash class retentiveness) as in IMS. Affinity-based routing was also found to allow for a considerably reduced number of global lock requests and fewer global lock conflicts. In [Yu87] the use of a central lock engine for concurrency control was additionally studied. Since the lock engine was assumed to provide very fast lock service at negligible overhead, this approach was found to support a higher number of nodes to be effectively coupled than with pass-the-buck.

The simulation study in [B88] compared the performance of four locking protocols for database sharing: a central locking scheme, two variations of primary copy locking and a disk controller locking as in TPF (see 3.1.1). The use of a sole interest concept or read optimization has not been considered, but message batching was assumed to be the main method to reduce the communication overhead. For primary copy locking, the fraction of remote lock requests was provided as a parameter. Disk controller locking was found to provide the best performance since every lock request could be combined with the disk read without extra overhead (this was due to an oversimplified buffer model without buffer hits). Similar throughput results were predicted for a central lock manager scheme if high batching degrees are applied. To keep response times sufficiently short despite the delayed propagation of lock requests, the use of intra-transaction parallelism was considered essential for the central locking scheme. The performance of database sharing with central locking was also compared with the performance of "shared memory" and "shared nothing" systems with the same number of processors. Despite the use of the simple concurrency control method, better throughput results were generally predicted for database sharing compared to database partitioning.

A simple analytic model was used in [IK88] to estimate the number of messages per lock request for two distributed locking schemes based on a fixed or dynamic GLA assignment (3.1.1). In the case of the fixed GLA assignment, a uniform distribution of lock requests among all nodes was assumed. The model ignored lock contention and assumed a random routing of the transaction workload. It was found that a dynamic GLA assignment can support a lower number of remote lock requests if there is locality of reference within transactions and if GLA assignments are based on a sufficiently small granularity.

In [Ra88a,b], a trace-driven performance evaluation of four concurrency/coherency control protocols was presented for six different workloads (the summary paper [Ra88b] discusses simulation results for two workloads). The best performance was generally observed for a primary copy locking protocol using read authorizations. Typically, this protocol required less than half the number of global lock requests than a central locking scheme using write and read authorizations at the page level. While the sole interest concept (write authorizations) helped to reduce the number of global lock requests, this technique was far less effective than the use of a local GLA in the primary copy scheme. Supporting node-specific locality by an affinity-based routing was found to support a much lower number of global lock requests than for random routing. Read authorizations proved to be very effective for the considered workloads and could limit the dependencies on the achievable amount of node-specific locality. The best optimistic protocol was a central validation approach combined with a preclaiming of locks for failed transactions (see 3.2). While the lock preclaiming ensured that a transaction is aborted at most once, the number of transaction rollbacks was still unacceptably high for workloads with a higher degree of update activity. The communication overhead for validation was generally lower than for the concurrency control messages in the pure locking protocols, in particular for a higher number of nodes. All protocols had severe performance problems with hot spot pages, e.g., for free space administration, if access to them was synchronized at the page level (in some cases, throughput even degraded when increasing the number of nodes). This shows the necessity to support record-level concurrency control or/and tailored protocols on such data objects.

The simulation study [Ra93a] has compared the primary copy protocol with the use of a global lock table that is maintained in the global memory of a closely coupled database sharing system. Since the latter approach allowed acquisition of a global lock within a few microseconds, substantially better performance could be obtained than with PCL. This was particularly the case for real-life workloads (represented by database traces) for which a high degree of node-specific locality was difficult to achieve for a higher number of nodes. For the debit-credit workload used in the TPC-A and TPC-B benchmarks [Gr91], the primary copy approach achieved comparable performance. For this workload the database and load can easily be partitioned so that almost all locks could locally be acquired for PCL.

4. Coherency Control

Coherency control techniques for database sharing basically have to solve two problems. First, invalidated pages in the database buffers must either be detected or avoided. Second, an update propagation strategy must be supplied that provides transactions with the most recent versions of database objects (pages). For both subproblems, we provide a classification and overview of the major solutions. The complete spectrum of coherency control schemes is obtained by combining the various alternatives for both subproblems with each other. In Subsection 4.1, we first describe the approaches for detection or avoidance of buffer invalidations. Update propagation strategies are outlined in Subsection 4.2. The major combinations are then summarized in Subsection 4.3.

There are close dependencies between coherency control and concurrency control for database sharing. In particular, there are different coherency control requirements and solutions for locking and optimistic concurrency control. For locking protocols it should be ensured that only current database objects are accessed. This can be achieved by suitable extensions of the locking protocol because an appropriate lock must be acquired before every object access. As we will see, with these *integrated concurrency/coherency control* schemes extra messages for coherency control can be avoided to a large extent. In OCC schemes, on the other hand, object accesses occur unsynchronized so that access to obsolete data generally cannot be detected before validation at the end of a transaction. To limit the number of validation failures and transaction aborts for OCC, it is essential that updates of committed transactions become quickly visible in all processing nodes.
Coherency control also depends on the data granularity for concurrency control. Page-level concurrency control simplifies coherency control since pages are the allocation units in the database buffers and transfer units between main memory and disk. Record-level concurrency control, on the other hand, would permit the same page simultaneously be modified in different nodes. To simplify the presentation, we assume page-level concurrency control in the first three subsections. In Subsection 4.4, we discuss the extensions required for record-level locking as well as for supporting multi-version concurrency control.
Performance studies on coherency control are summarized in 4.5.

4.1 Buffer invalidation detection and avoidance strategies

There are four major alternatives for detection or avoidance of buffer invalidations in database sharing systems: broadcast invalidation, on-request invalidation, buffer purge and retention locks. Broadcast invalidation is applicable for any concurrency control method, but introduces the greatest overhead. The other approaches can only be used in combination with locking protocols. The next two subsections describe the broadcast and on-request invalidation approaches to detect buffer invalidations, respectively. Avoidance strategies (buffer purge, retention locks) are covered in Subsection 4.1.3.

4.1.1 Broadcast invalidation

In this approach, a broadcast message is sent at the end of an update transaction indicating the pages that have been modified by the respective transaction. The broadcast message is used to detect the invalidated page copies in other nodes and to immediately remove them from the database buffers thereby preventing access to them. To reduce the communication overhead, multiple broadcast messages of different update transactions of a node may be bundled together at the expense of a delayed message transmission and invalidation. Such an optimization is similar to the group commit technique to reduce the number of log I/Os [Ga85].

For locking schemes it is required that the obsolete page versions are discarded from all buffers before the update transaction releases its write locks (otherwise, access to obsolete data would be possible). Therefore, the lock release must be delayed until all nodes have acknowledged that they have processed the broadcast message and discarded the invalidated page copies. Since update transactions must synchronously wait for all acknowledgment messages, we call this approach ***synchronous broadcast invalidation***. A major disadvantage is that response time and lock holding time are significantly increased. Furthermore, a substantial communication overhead is introduced that grows with the number of systems. Despite these problems, some existing database sharing systems use synchronous broadcast invalidation (e.g., IMS).

A variation of synchronous broadcast invalidation called *selective notification* has been proposed in [DY91]. It assumes a GLM-based locking scheme where the GLM keeps track of every node's buffer contents. This information can be used to restrict invalidation messages to those nodes that actually hold copies of the pages modified by a transaction. A list of these nodes can be obtained from the GLM together with the write lock thereby avoiding extra messages. Changes in the local buffer state can also be communicated to the GLM without extra messages, in general, by piggy-backing this information with lock request and release messages. Still, the maintenance overhead at the GLM may be substantial since the global lock table has to be updated for every page that is newly cached or replaced in any node.

For OCC, access to invalidated data is detected during validation. However, to reduce the number of validation failures broadcast invalidation can be used to remove obsolete pages from the buffers. In this case, an ***asynchronous broadcast strategy*** is sufficient where the invalidation message is sent after the end of an update transaction. As a result, no acknowledgment messages are required and response times are not increased by the broadcast invalidation. As pointed out in [Ra87b], the broadcast messages can also be used to immediately restart the transactions that have already accessed the invalidated objects and are thus doomed to fail during validation. The early abort of these transactions saves unnecessary work for their completion and validation.

4.1.2 On-request invalidation

This approach, also referred to as "check-on-access", is applicable for the locking protocols using a GLM (central locking or distributed locking with fixed or dynamic GLA assignment). The idea is to use extended information in the global lock table which allow the GLM to decide upon the validity of a buffer (cached) page together with the processing of lock requests. Since a lock has to be acquired before a (cached) page can be accessed, obsolete page copies can be detected without any additional communication and response time increase, a main advantage compared to broadcast invalidation schemes. The information needed to detect buffer invalidations is also updated without extra communication together with the release of the write lock. A disadvantage compared to a broadcast invalidation is that obsolete pages are detected and removed later from the buffer so that hit ratios may be worse.

In [Ra86], two methods for on-request invalidation are described. One approach to detect buffer invalidations is the use of ***page sequence numbers*** that are stored in the page header and incremented for every modification. When the write lock required for a modification is released at the GLM, the current value of this sequence number is stored in the global lock table. When a lock is requested at the GLM, it is first checked whether the corresponding page is cached in the local buffer and, if so, what the sequence number of the cached copy is. The GLM can then easily determine whether the cached page is obsolete so that access to invalidated data can be avoided. Such an on-request invalidation scheme based on page sequence numbers is used in DEC's database sharing systems for VaxClusters [KLS86] and in [MN91].

In [Ra86], we have described an alternative using so-called ***invalidation vectors*** for detecting buffer invalidations. For every modified page, an invalid bit is used per node to indicate whether the respective node *may* hold an obsolete version of the page. When a write lock is released, the GLM sets this bit for all nodes except for the node where the modification was performed. When a lock is requested and a copy of the respective page is cached, the bit indicates whether the copy is up-to-date. The invalidation vector technique has the advantage that it does not depend on sequence numbers stored with the object so that it could be used to detect invalidations for objects of arbitrary size (e.g., records). Such an approach would be especially valuable for object-oriented DBMS that support object caching rather than page caching. In 4.2.2, we discuss an example that is based on the use of invalidation vectors (Figure 6 and Figure 8).
The on-request invalidation scheme in [DIRY89] requires maintenance of the complete buffer state at the GLM. It therefore introduces a substantially higher processing and space overhead than a sequence number or invalidation vector approach.

On-request invalidation also works when read or/and write authorizations are used to save global lock requests. This is because as long as these authorizations are held no other node can modify and therefore invalidate the respective page(s).

Coherency control information for on-request invalidation must be kept even when there are no current lock requests for the respective pages. To limit the size of the global lock table, it becomes necessary to periodically delete global lock table entries for pages that have not been referenced for longer periods of time. However, since this can result in the loss of relevant coherency control informations special precautions are necessary to ensure that no invalidated page copies will be accessed later on. There are two alternatives for this purpose:

- Before an entry with coherency control information is discarded a broadcast message is sent to all nodes so that they can remove invalidated copies of the respective page. Such broadcast messages can be bundled together and sent asynchronously. If invalidation vectors are used to detect buffer invalidations, only those nodes need to be informed for which the invalid bit indicates a possible invalidation.
- The broadcast messages are avoided if a cached page is only used when there is an entry for the page in the global lock table. With this approach it must only be ensured (by the update propagation strategy, see 4.2) that the page copy on disk is up-to-date before an entry is deleted from the global lock table.

4.1.3 Avoidance of buffer invalidations

Buffer invalidations are only possible for cached pages that are modified at another node. While a cached page is locked by an active transaction, it is protected against remote modifications and thus cannot get invalidated. Consequently, buffer invalidations are avoided altogether if pages are purged from the database buffer before the lock release at EOT. Such a ***buffer purge*** approach, however, is of little relevance since it implies a FORCE strategy for modified pages and poor hit ratios since inter-transaction locality cannot be utilized any more to save disk reads.

A better approach is to retain the pages in main memory but to protect them from invalidation by special ***retention locks*** [HR85, Ra88a]. Consequently, for every cached page either a retention lock or a regular transaction lock must exist. Before a page can be modified at another node, a write lock must be requested which will conflict with either the regular or the retention lock. Before the transaction lock or retention lock is released and the write lock is granted to the other node, the corresponding page is purged from the buffer so that its invalidation is avoided.

Such an approach is particularly attractive for the locking protocols using read and write authorizations. This is because retention locks can be implemented by read and write authorizations at the page level resulting in two types of retention locks (WA and RA). For modified pages not currently locked by an active transaction, a WA retention lock must be held guaranteeing that no other system holds a lock or retention lock (copy) for that page. This exclusive retention lock permits a local synchronization of read and write locks (write authorization). Unmodified cached pages are protected by a RA retention lock which can be held by multiple nodes concurrently. In addition, RA guarantees that no node holds a write lock or WA retention lock thus permitting a local synchronization of read accesses (read authorization). In 4.2.2, we further describe this approach when discussing update propagation schemes.

In [MN92a], a similar scheme as the one sketched above is described. They differentiate between "physical locks" requested by the buffer manager to avoid buffer invalidations and "logical locks" requested by the lock manager on behalf of transactions. Physical locks are used in the same way as retention locks. In particular for every cached page a physical lock is required in shared or exclusive mode which is also used by the lock manager to locally process logical lock requests.

4.2 Update propagation

Basically two forms of update propagation for pages need to be considered in database sharing systems. ***Vertical update propagation*** refers to the propagation of modified pages between main memory buffers and disk. In addition, a ***horizontal update propagation*** is required to exchange pages (modified database objects) between nodes.

For vertical update propagation, we have the same alternatives as in centralized DBMS, namely the FORCE or the NOFORCE approach[5] [HR83]. A FORCE strategy for update propagation requires that all pages modified by a transaction are forced (written) to the permanent database on disk before commit. This approach is usually unacceptable for high performance transaction processing since a high I/O overhead and significant response time delays for update transactions are introduced. On the other hand, there is no need for redo recovery after a node failure. Furthermore, FORCE simplifies coherency control for database sharing since it ensures that the most recent version of a page can always be obtained from disk. Hence, no extra support for horizontal update propagation is needed since all modifications are implicitly exchanged across the shared disks. Coherency control merely consists of one of the approaches discussed in 4.1 to detect and discard or to avoid invalidated pages cached in main memory. The rest of this subsection will therefore concentrate on the NOFORCE alternative.

NOFORCE permits a drastically reduced I/O overhead compared to FORCE and avoids the response time increase due to synchronous disk writes at EOT (end of transaction). Only redo log data is written to a log file at EOT, and multiple modifications can be accumulated per page before it is eventually written to disk. Since the permanent database on disk is not up-to-date, in general, coherency control has to keep track of as to where the most recent version of a modified page can be obtained. Instead of reading a page from disk, ***page requests*** may have to be sent to the node holding the current page copy in its buffer.

One approach for horizontal update propagation would be to broadcast a modified page to all nodes at EOT (or multicast the page to the nodes where the old version is cached) [Yu87]. In this case invalidated pages could not only immediately be removed from the buffers (as in the broadcast invalidation scheme) but the new page version could directly be installed thereby improving hit ratios. The enormous communication overhead and bandwidth required by such a strategy seems not practical, however. Even in this case, pages may have to be requested from other nodes since not all

5. This distinction is analogous to the "write-through" (FORCE) vs. "write-back" (NOFORCE) dichotomy for update propagation between processor caches and main memory [Sm82].

modified pages can be cached in all buffers (unless we have a completely replicated main memory database).

More practicable approaches for horizontal update propagation (NOFORCE) assume that one of the nodes is the ***owner*** for a modified page (i.e., a page for which the disk version is obsolete). The page owner has two major responsibilities. First, it provides other nodes on request with the current page version. Furthermore, only the owner is generally allowed to write out the page to disk. The latter regulation avoids an extra synchronization to ensure that no two nodes try to concurrently write out the same page or to avoid that the current version of a page on disk is overwritten by an obsolete copy. Note that no page owner is necessary for pages whose current version resides in the permanent database on disk (in this case, the disk could be viewed as the page owner). In the case of FORCE, no extra synchronization for disk writes is required if transactions acquire exclusive page-level locks on modified pages since these locks are held until all force-writes are completed.

Assuming an owner-based approach for horizontal update propagation, update strategies for NOFORCE can be classified according to two aspects. The first criteria is concerned about how pages are physically exchanged between nodes (e.g., across the communication system or over the shared disks); these alternatives will briefly be discussed in 4.2.1. The second classification criteria distinguishes between different page ownership strategies (Subsection 4.2.2). Here, similar approaches as for the allocation of global lock authorities (Section 3.1) are applicable (central, distributed fixed or distributed dynamic page ownerships). The resulting alternatives for update propagation are summarized in Figure 5.

4.2.1 Page transfer schemes

A page can be transferred from the owner to a requesting node either directly across the communication system ("memory-to-memory" transfer), across the shared disk or across a shared intermediate memory. A page exchange across disks incurs the longest delay since it requires two synchronous disk I/Os (30-50 ms) and two messages (the page is requested from the owner by a page request message; after the owner has written out the page it

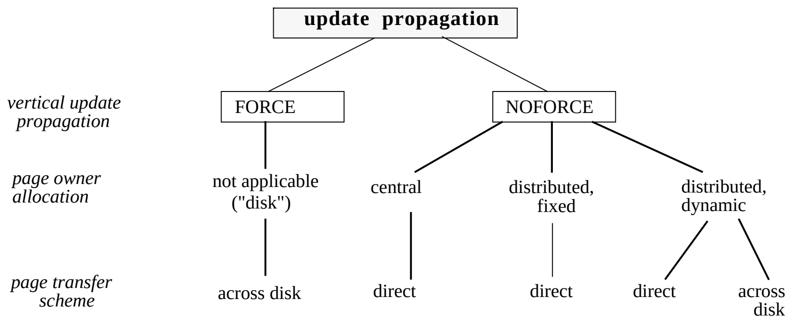


Figure 5. Update propagation strategies

sends back a response message indicating that the page can be read from disk). If the disks are equipped with a non-volatile disk cache, the delay can be substantially improved to about 3-8 ms for two I/Os. A direct page exchange, on the other hand, avoids the I/Os altogether and sends back the page within the response message for a page request. With a fast interconnect, the transmission delay should be in the order of 1 ms plus the CPU time for sending and receiving the two messages. A shared intermediate semiconductor memory with access times of 10-50 microseconds per page could support even faster transfer times [Ra91].

In centralized DBMS and database sharing systems using FORCE, a buffer miss results in an I/O to read the respective page from disk. For database sharing and NOFORCE, a page request with an exchange of the page across disks more than doubles the delay and overhead. A direct exchange of pages over the network, on the other hand, permits a shorter delay than to read the page from disk, but the overhead for the page request and response messages is typically substantially higher (2 send and 2 receive operations) than for a single I/O.

The method used for exchanging modified pages also has a significant impact on crash recovery [Ra89, MN91]. If modified pages are always exchanged across the shared disks (and a page is not permitted to be concurrently modified in different nodes), crash recovery is simplified since it can be done with the local log file of the crashed node. This is because all modifications of other nodes are already reflected in the permanent database so that at most updates of the crashed node may have to be redone (or undone). On the other hand, with a direct exchange of modifications a page can be modified at multiple nodes before it is written to disk. Hence, redoing modifications of a crashed node may require the use of redo information from remote nodes. To apply the redo information in correct order, the local log information may have to be merged in a global log[6]. A page exchange across a shared intermediate semiconductor memory may have the same recovery implications as a page transfer over the network or across disks, depending on whether or not the memory is volatile.

Depending on the page ownership strategy, a page may be modified at a node different from the node holding the page ownership. As we will see, this is possible for the central and distributed fixed page ownership schemes (for distributed dynamic ownerships the node performing the modification becomes the new page owner). In these cases, page transfers are not limited to pages requested *from* the owner, but pages may also have to be transferred *to* the owner. For these page transfers, only a direct page transfer over the communication network (or intermediate memory) is meaningful. Therefore, a page transfer across disk is limited to schemes with a dynamic page ownership allocation (Figure 5).

6. As a compromise between a page exchange across disks and a direct page transfer, [MN91] also considered a "medium scheme" that still supports the simpler crash recovery. In this approach the owner writes a page synchronously to disk upon a page request but concurrently sends the page to the requesting node. If the message transfer is successful (the normal case), the I/O delay for reading the page from disk is saved at the expense of an additional page transfer message compared to the page exchange across disks.

4.2.2 Page ownership strategies

We distinguish between a central approach where one node is the owner for all pages and distributed schemes. In the latter case we differentiate between a fixed and dynamic assignment of page ownerships. In general, we assume a direct exchange of pages over the network; in the case of dynamic page ownerships the alternative of exchanging modified pages across disk will also be considered.

Central page ownership

In this case a single node holds the ownership for all modified pages. As a result, copies of all pages modified by a transaction must be sent to the central page owner node at EOT. The scheme is similar to FORCE with the difference that pages are sent over the network to a single node rather than written to disk. While these page transfers may be faster than the disk I/Os, the overhead for sending the pages is probably higher than for the disk writes. What is more, the central node is an obvious performance bottleneck since it has to receive a high number of pages and to perform all disk writes.

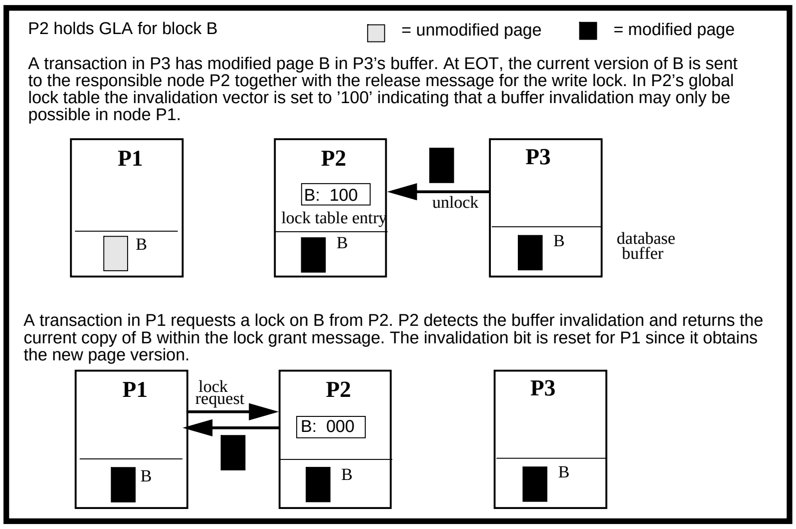
On the positive side, page requests and transfers can be combined with concurrency control messages if the central node is additionally used for global locking (centralized locking approach). Page requests can then be forwarded to the central server together with lock requests. The page itself is returned to the requesting node together with the lock grant message, provided the page is still cached at the central node (otherwise it can be read from disk by the respective node). Finally, the page transfers to the central server can be combined with the messages for releasing the write locks at EOT.

The total number of page transfers to the central page owner node may significantly be reduced when a central locking protocol with write authorization is used. This is because a write authorization permits a node to modify a page multiple times without requesting or releasing a write lock at the global lock manager. Hence, the page is not transferred to the central node until the write authorization is explicitly revoked (due to a lock conflict with other nodes) or voluntarily been given up. Even with such an optimization, however, the central page owner (global lock manager) node remains a likely bottleneck when high transaction rates are to be supported.

Distributed fixed page ownerships

The danger of a central performance bottleneck can easily be reduced by distributing the page ownerships to multiple nodes. In the case of a fixed ownership allocation, this assignment is known to all nodes so that no extra messages are needed to locate the page owner. For determining the page ownership allocation, the same methods as for a GLA (global lock authority) allocation can be used, e.g., by using a hash function or a logical partitioning of the database (3.1). The use of a logical partitioning as in the primary copy protocol seems most attractive since it supports an affinity-based routing which helps to reduce the number of page transfers. In this case every node holds the page ownership for one logical database partition. For all page accesses on the local partition a page request message is saved; similarly, modifications of pages of the local partition save the page transfer message to the owner.

Such a page ownership strategy could be used in combination with virtually all concurrency control strategies[7]. However, the most efficient solutions result if such an ownership approach is used in combination with a distributed locking scheme with fixed GLA assignment like primary copy locking, where the GLA and page ownership allocations are identical. This is because extra messages for coherency control may be completely avoided, similar as sketched for the central ownership approach. In particular, page requests can always be combined with lock requests to the authorized node acting as both the global lock manager and the page owner. Furthermore, page transfers to the page owner can be combined with the lock release (unlock) message and pages may be returned to the requesting node together with the lock grant message (alternatively, the response message indicates to read the page from disk if the page owner has not cached the respective page). Finally, extra messages for detecting buffer invalidations can be avoided by employing an on-request invalidation scheme at the global lock manager. As a result, no extra messages for coherency control are necessary. An added benefit is that buffer invalidations are limited to pages from non-local partitions.



A transaction in P3 has modified page B in P3's buffer. At EOT, the current version of B is sent to the responsible node P2 together with the release message for the write lock. In P2's global lock table the invalidation vector is set to '100' indicating that a buffer invalidation may only be possible in node P1.

A transaction in P1 requests a lock on B from P2. P2 detects the buffer invalidation and returns the current copy of B within the lock grant message. The invalidation bit is reset for P1 since it obtains the new page version.

Figure 6. Example for primary copy locking with fixed page ownerships, on-request invalidation based on invalidation vectors and direct exchange of modified pages

7. For instance, in [BHT90] a "partitioned optimistic" concurrency control scheme is discussed which is based on a central validation strategy and distributed fixed page ownerships according to a logical partitioning.

Figure 6 illustrates such an approach for primary copy locking with a combined allocation of the GLA and page ownerships. Buffer invalidations are detected by an on-request invalidation scheme using invalidation vectors (4.1). The example shows a "worst-case" scenario with respect to message traffic since both accesses to page B occur outside the owning processing node P2. Coherency control is performed without additional messages; in particular, the page transfers are combined with the lock release and grant messages. Note that the invalid bit is set to "0" (false) not only for the node where the modification has been performed (P3) but also for the page owner (P2) since it obtains the current page version before the write lock is released.

In terms of number of messages, the sketched coherency control approach based on a combined fixed assignment of GLA and page ownerships is optimal. In particular, separate messages and deactivations for page requests are avoided. On the negative side, many pages may have to be sent to the page owner (together with the lock release) without experiencing a re-reference before being written to disk. Although these transfers do not require an extra message, the communication overhead and bandwidth requirements for these long messages are of course higher than for short lock release messages. An affinity-based transaction routing aiming at a workload allocation such that most database accesses occur at the owner node can significantly reduce the number of page transfers.

Distributed dynamic page ownerships

In this approach, the node performing the last modification of a page becomes its owner. As a result, the page ownership dynamically migrates between nodes according to the distribution of update requests. An advantage of this approach is that no messages are needed to send modified pages to a predetermined page owner at EOT. On the other hand, a locating method is required to determine the current page owner in order to obtain the most recent page version.

To locate the current page owner, similar methods as discussed for distributed locking protocols with a dynamic GLA assignment can be used (3.1.1). In particular, a directory could be used that is replicated in all nodes or partitioned according to a hash function. Here, the distributed hash table approach is less desirable since it would introduce extra messages to locate the page owner in addition to the messages for the page requests. The use of a replicated locating directory, on the other hand, is quite appropriate for coherency control schemes based on a broadcast invalidation (e.g., for optimistic concurrency control). In this case, the replicated directory can be maintained with little extra overhead since the broadcast messages used to detect invalidated pages already indicate the page owner (i.e., the node where the modification has been performed) [Ra87a,b].

For locking protocols supporting the notion of a global lock manager, an even more efficient locating strategy is feasible by recording page ownerships in the global lock table. This approach incurs no replication overhead and also avoids extra messages to locate the current page owner since it can be determined during lock processing. The adaptation of the page ownership information also occurs without extra messages during the processing of write locks. While extra messages for locating the owner can be avoided, the page requests themselves may cause messages and deactivations in addition to those for the lock requests (since the owner is not known until the lock request has been processed). After the owner has written the page to disk, it informs the GLM (by an asynchronous message) to reset the page ownership field in the global lock table indicating that the current page version can be read from disk. This helps to largely avoid page requests that can no longer be satisfied by the former page owner.

The dynamic page ownership approach can be used in combination with all concurrency control methods from Section 3 and with all three techniques to detect or avoid buffer invalidations (4.1). Since it is impossible to discuss all possible combinations in more detail, we concentrate on three cases that appear most relevant.

(1) *Central lock manager + retention locks + dynamic page ownerships*

As discussed in Subsection 4.1.3, two types of retention locks corresponding to a read authorization (RA) and write authorization (WA) can be used to avoid buffer invalidations and to support a local processing of lock requests. Buffer invalidations are avoided since 1) for every cached page a transaction lock or a retention lock must be locally held which conflict with external write lock requests and 2) a page is purged from the buffer before a node's RA or WA retention lock (or the last transaction lock) is released. For the assignment of page ownerships we distinguish between two subcases depending on whether modified pages are exchanged across the disks or directly over the network.

a) Page exchange over disks

In this case, the page owner corresponds to the node where a write transaction lock or a WA retention lock is held. Since the GLM records the mode of granted locks in the global lock table, the page owner is known without having to maintain additional information. When a modified page exists in the system, it is ensured that the page is only cached at the page owner's node due to the compatibility of lock modes. In the scenario shown in Figure 7a, node P1 is the owner for page B since it holds the write authorization. When another node P2 wants to access the page, the lock conflict is detected at the GLM which results in a combined revocation request and page request to the current page owner (lock holder). Before the WA retention lock is released, the page owner P1 writes the modified page to disk (step 3). After the GLM has granted the lock to P2, P2 can read the page from disk (exchange of modified pages across disk). If P2 has requested a write lock, P1 completely gives up its retention lock and purges the page from the buffer; P2 becomes the new page owner. If a read lock was requested by P2, P1 can keep the page in its buffer and retain a RA retention lock. In this situation, there is no page owner in the system any more since the current page version can be read from the permanent database due to the page exchange across disk.

Note that with this approach extra messages for page requests are avoided since they can be combined with the revocation of write authorizations. Furthermore, no page transfer messages occur since pages are exchanged across disk. A RA retention lock always ensures that the most recent page version resides in the local buffer or can be found on disk. On the other hand, if a page is replaced from the database buffer due to normal replacement decisions, this indicates that it has not been referenced for some time. Therefore, it is advisable to release the associated retention lock voluntarily in order to limit the number of revocations and lock table entries.

Oracle uses such a coherency control scheme [Or90]. In [MN92a], a variation is proposed where the page owner sends a modified page to the requesting node concurrently with the disk write (step 3); the write lock (WA retention lock), however, is still held until the disk write is completed. This has the advantage that by the time the requesting node obtains its lock (step 5) it has already received the page in most cases so that the I/O delay to read in the page (step 6) is avoided.

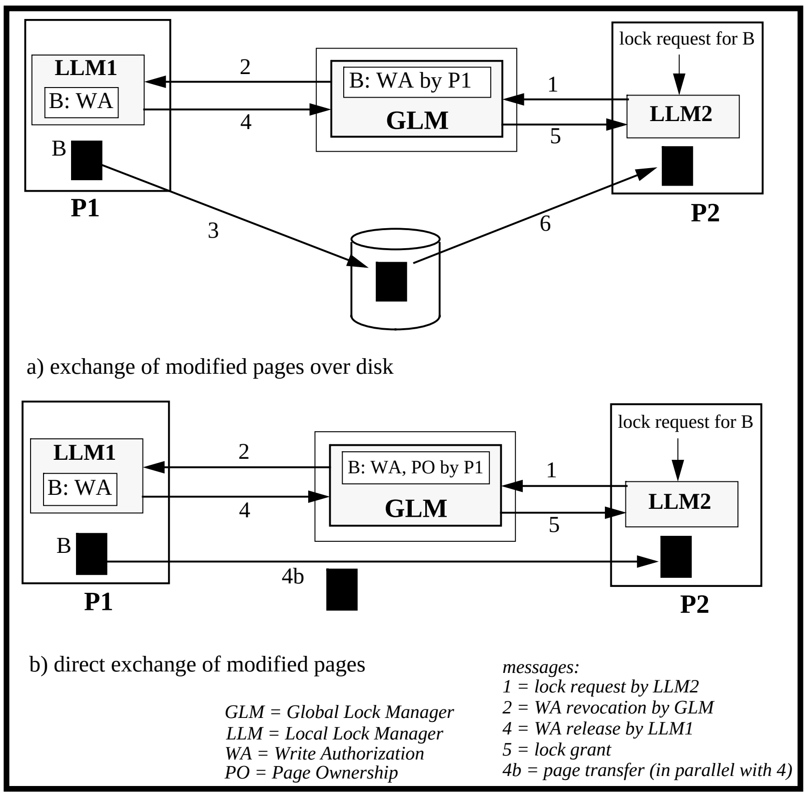


Figure 7. Update propagation with retention locks

b) Direct page exchange over network

The main difference compared to the previous case is that when a WA retention lock is revoked, the page is not written out by the page owner before the lock is released. Rather, the page is directly sent to the requesting node concurrently with the release message sent to the GLM (Figure 7b). Thus lock acquisition and page exchange are significantly faster than before since two disk I/Os (steps 3 and 6) are avoided.

Due to the direct exchange of modified pages it is necessary to keep the page ownership even in the case when no write lock or WA retention lock is granted. For instance, if a read request caused the WA revocation, the WA retention lock is downgraded to a RA retention lock. However, the page owner now has to keep its page ownership since the page has not yet been written to disk. Furthermore, other nodes that may wish to access the page have to know from where they can get the current page version. Therefore, the page owner has to be recorded in an additional field in the global lock table. Note that in the case of a RA retention lock and a local buffer miss, a page request to the page owner is necessary to obtain the current page version. Again, it is therefore recommended to release a read authorization when the corresponding page is being replaced.

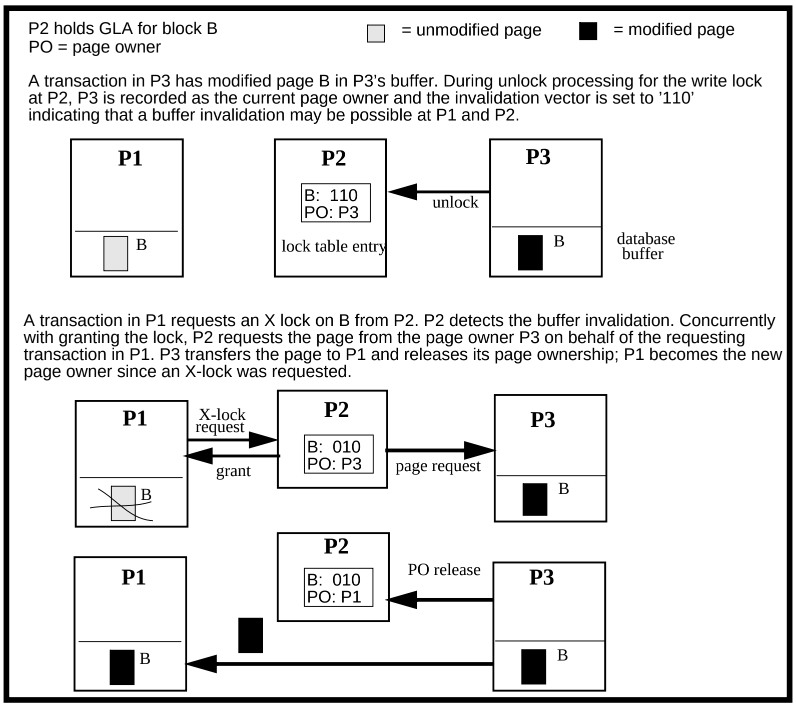
The ownership for a page migrates as soon as there is a write request by another node. When the new owner node already holds a current version of the page, e.g., due to a preceding read request, the ownership is transferred without page transfer.

(2) *Primary copy locking + on-request invalidation + dynamic page ownerships*

In contrast to the fixed page ownership allocation discussed above, now the modified page needs not be sent to the GLM node together with the release of the write lock. During unlock processing for a write lock, the responsible GLM merely records in the global lock table where the page ownership currently resides (i.e., where the last modification of the page was performed). When a lock is granted to a transaction, the GLM indicates in the grant message which node is the current page owner. The page is then requested from the owner by a separate message. Alternatively, the GLM could directly forward the page request to the page owner on behalf of the requesting node to speed up the page transfer. Of course, if the page is already cached at the requesting node and still valid this copy can be used. Furthermore, the page can be read from disk if the GLM indicates that there is no current page owner.

Figure 8 illustrates the use of dynamic page ownerships for primary copy locking for the example already used in Figure 6. The release of the write lock for page B by P3 is now a short message to node P2. Before releasing the write lock, the GLM in P2 records P3 as the current page owner in its global lock table[8]. Furthermore, the invalidation vector is now set to "110" rather than "100" since there may be a buffer invalidation in node P2. In the lock grant message for the transaction in P1, it is indicated that the local copy is invalid and that the page is to be requested from P3. The page

8. Alternatively, the page ownership could already be assigned when the write lock is granted.



P2 holds GLA for block B
PO = page owner
= unmodified page
= modified page
A transaction in P3 has modified page B in P3's buffer. During unlock processing for the write lock at P2, P3 is recorded as the current page owner and the invalidation vector is set to '110' indicating that a buffer invalidation may be possible at P1 and P2.
P1
P2
P3
B: 110
PO: P3
unlock
lock table entry
B
database buffer
A transaction in P1 requests an X lock on B from P2. P2 detects the buffer invalidation. Concurrently with granting the lock, P2 requests the page from the page owner P3 on behalf of the requesting transaction in P1. P3 transfers the page to P1 and releases its page ownership; P1 becomes the new page owner since an X-lock was requested.
X-lock request
grant
B: 010
PO: P3
page request
PO release
B: 010
PO: P1

Figure 8. Example for primary copy locking with dynamic page ownerships, on-request invalidation based on invalidation vectors and direct exchange of modified pages

request is directly performed by the GLA node which also requests P3 to give up its page ownership since an X lock has been requested at P1. After P3 has released the page ownership, P1 is kept as the new page owner in P2's lock table. If P1 had requested a read lock, the page transfer would have been performed in the same way but without page ownership migration. If the last modification had occurred at the GLM node P2, the page could have been returned to P1 within the lock grant message thereby avoiding the separate page request.

(3) Dynamic GLA assignment + dynamic page ownerships

Such a combination results in a similar approach as in the previous two cases depending on whether an on-request invalidation scheme or retention locks are used to detect or avoid buffer invalidations. Compared to case 1, the main difference is that not only the page owner but also the global lock manager for a page may dynamically migrate. We assume that coherency control information, including the page owner, is stored in the global lock table[9]. This implies that the GLA cannot be released after the last lock for the page has been released, but that it must be retained as long as a page

9. The page owner could also be recorded in the locating table (e.g., distributed hash table) used to determine the global lock manager for a page.

owner exists (i.e., the page is modified and not yet written out) and other coherency control information to detect invalidated pages has to be maintained.
Note that now six messages may be needed to acquire a lock and the corresponding page when GLM and page owner reside in different nodes (two for determining the GLM node, two for acquiring the lock, and two for obtaining the page from the page owner). To avoid this worst case, one could try to allocate GLM and page owner to the same node by changing the GLA assignment together with the page ownership. That is, whenever the ownership of a page migrates to another node (due to a write request) the GLA for the respective page also migrates to that node. In this case, the worst case can be limited to four messages for acquiring a lock and the page. On the other hand, migration becomes more expensive since the locating information for finding the global lock manager has to be adapted potentially requiring extra messages.

4.3 Combining the concepts

During the previous subsections, it was already necessary to consider various combinations of the techniques to detect or avoid buffer invalidations and for update propagation. In Table 3 the main combinations are summarized together with a qualitative assessment of their relative performance. Not shown are the alternatives for exchanging modified pages between nodes in the case of NOFORCE. From the performance point of view, a direct exchange over the network (or a fast intermediate memory) is clearly the preferred method. An exchange of modified pages across disks (which simplifies recovery) could be fast enough for high-performance requirements when disks with non-volatile disk caches are used. Also not shown in Table 3 are the many possible combinations with different concurrency control protocols. Since these combinations cannot easily be presented in a table or diagram, we discuss the major dependencies and recommendations in the text.

The weakest ratings in Table 3 are given for schemes based on a FORCE strategy (high I/O overhead and delays), for a central page ownership approach (high communication overhead for page transfers, central bottleneck) and for broadcast invalidation (high communication overhead, response time increase in the case of synchronous broadcast invalidation). FORCE as well as a broadcast invalidation can be combined with any concurrency control scheme; the central page ownership scheme

detection / avoidance scheme — page ownership	broadcast invalidation	on-request invalidation	retention locks
disk (FORCE)	--	-	-
central (NOFORCE)	--	-	-/o
fixed distributed (NOFORCE)	-	+	o
dynamic distributed (NOFORCE)	-	+	+

- /-- not recommended / poor performance
O not recommended / medium performance
+ recommended / good performance

Table 3. Combination of coherency control strategies

could be used in combination with a central locking or central validation protocol.
For NOFORCE with a fixed assignment of page ownerships, the most efficient solutions are based on distributed locking schemes with a corresponding fixed GLA allocation (e.g., primary copy locking). In this case, an on-request invalidation scheme is most appropriate for detecting buffer invalidations since it supports coherency control without any extra messages. A dynamic page ownership approach can be used in combination with all concurrency control schemes. Here, the use of retention locks to avoid buffer invalidations is particularly appropriate for a central locking scheme, but can also be used for protocols like pass-the-buck [HR85]. For distributed locking protocols with a global lock manager, on-request invalidation seems most attractive. Optimistic concurrency control protocols can also employ a dynamic page ownership approach, but they are limited to an (asynchronous) broadcast invalidation.

The description in 4.1 and 4.2 has shown that extra messages for coherency control can be avoided to a large extent, in particular for concurrency control algorithms based on locking. Even in the case of NOFORCE, page requests and page transfers can frequently be combined with regular concurrency control messages. Separate page request messages are primarily needed for dynamic page ownership schemes. If retention locks are used to avoid buffer invalidations, however, page requests can often be combined with the revocation of write authorizations. In all coherency control schemes, the number of buffer invalidations, page requests and page transfers strongly dependens on the locality of database access to be supported by an affinity-based transaction routing.
Appendices A and B summarize which concurrency and coherency control protocols are used in some existing database sharing systems or have been described in the literature.

4.4 Support for reduced data contention

The coherency control protocols described so far assumed a concurrency control at the page level. Although many existing DBMS only support page locking, additional provisions are generally necessary to keep data contention sufficiently low. To reduce data contention, commercial DBMS often offer the choice of a *reduced consistency level* ("cursor stability") instead of serializability ("repeatable read") [CLSW84, Ta89]. With cursor stability, data contention is improved by keeping only write locks until commit while read locks are merely held during the actual access by the transaction. This approach impairs consistency since a transaction may see different versions of an object due to concurrent modifications (unrepeatable read [GLPT76]). The cursor stability option can be used for database sharing without additional problems. However, since a transaction generally has to request more locks than with "long" read locks, more global lock requests may be needed. The read optimization can help to avoid extra messages since when a read authorization is assigned for the first read access of a transaction, subsequent read lock requests for the same object are likely to be locally processed.

In the following, we discuss two other approaches to reduce data contention namely support for multi-version and for record-level concurrency control. To limit the scope of our discussion, we only consider locking protocols in this section. Optimistic schemes have difficulties with record-level concurrency control even in the centralized case due to the fact that modifications have to be performed on private object copies [Hä84, Mo92].

4.4.1 Multi-version concurrency control

Multi-version concurrency control is a general approach to reduce data contention [Ch82, CM86, BHG87]. The basic idea is to provide read-only transactions with the database state that was valid at their BOT (begin of transaction); modifications that occur during their execution remain invisible to read-only transactions. Update transactions, on the other hand, always access the most recent database state and their modifications generate new object versions. The advantage of this approach is that concurrency control is only needed between update transactions thereby considerably improving data contention. Read-only transactions are never roll-backed or blocked due to data contention. This desirable property has been shown to substantially improve performance in centralized DBMS [CM86, HP87]. On the other hand, read-only transactions may see a slightly out-dated (but consistent) database state and multiple versions of database objects have to be maintained. Fortunately, older object versions have to be kept only for a limited time; they can be discarded as soon as the respective read-only transactions for which they are held have been committed.

Two major difficulties need to be addressed to support multi-version locking for database sharing: 1) determination of the appropriate object version for read-only transactions and 2) provision of the respective version. These problems are similar to the coherency control problem where we had to determine the validity of a page and to provide the most recent version of a page. To make the coherency control solutions applicable to multi-version concurrency control we assume that versions are maintained at the page level.
A simple method to select the correct version of a modified page is to assign a globally unique and monotonically increasing timestamp to committed update transactions and to the pages modified by them[10]. Furthermore, every read-only transaction obtains such a global timestamp at its BOT. A read-only transaction must then see the youngest object versions with a page timestamp that is smaller than its BOT timestamp. The information which page versions are available can be kept in the global lock table or in a replicated locating directory that is updated by broadcast messages sent during the commit of update transactions. To provide the correct page version, the same ownership strategies as discussed in Subsection 4.2.2 can be employed. In the case of a fixed page ownership allocation according to a primary copy scheme,

10. This may be achieved by a global hardware clock or perfectly synchronized local clocks. Other methods to determine object versions in a distributed environment (requiring extra messages) have been discussed in [CG85; We87].

every node maintains all versions for pages of its partition. With a dynamic ownership strategy, on the other hand, different versions of the same page may reside in different nodes; the respective page owners may also be recorded in the global lock table or in a replicated directory. Keeping the version information in the global lock table has the advantage that it can be maintained without extra messages during the lock processing for update transactions. Read-only transactions, however, now have to access the global lock table for every object access to obtain the correct page version although they do not have to request locks any more. More details on multi-version concurrency control for database sharing can be found in [Ra88a].

4.4.2 Record-level locking

Record-level locking permits different records of the same page be concurrently modified at different nodes. As a result, each node's page copy is only partially up-to-date and the page copy on disk at first contains none of the modifications (Figure 9). Since writing out partially up-to-date pages could lead to lost updates, all modifications must be merged before the page can be written to disk. This merging of record modifications is cumbersome, in general, and may cause a substantial amount of extra messages. Furthermore, even with record-level locking short page locks/latches may be needed in order to serialize modifications to page control information (e.g., free space information in the page header). For database sharing, this would require additional messages and seems impractical.

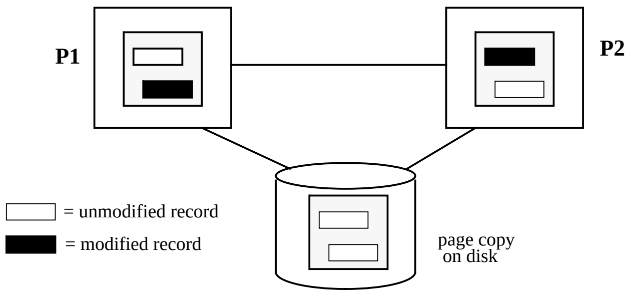


Figure 9. Coherency control problem with record-level locking

We discuss two restricted forms of record-level locking that avoid an explicit merging of modified page portions. At the end, we discuss a third possibility for a "full record-level locking" that is restricted to record updates that leave the page structure unchanged (e.g., no insertions or delete operations).

1. *Local record-level locking only*
   The simplest approach is to use record-level locking only within a node, but to resolve global lock conflicts at the page level. In this case, the global locking protocol and coherency control could remain unchanged; only the local lock manager needs to be extended compared to the pure page-level schemes. Such an approach is particularly at-

tractive for the locking schemes that utilize either a local GLA and/or write authorizations to locally process lock requests. While the GLA and the write authorizations are assigned at the page level, lock requests for which the GLA is local or for which a write authorization exists can be requested at the record level thereby reducing the number of local lock conflicts. Another significant benefit is that no additional messages are needed since remote lock requests are still at the page level. With an affinity-based routing that tries to maximize the usefulness of a local GLA and write authorizations, such a simple form of record-level locking can already support a substantially lower data contention without much overhead. Note that support of node-specific locality also means that most lock conflicts should occur between transactions of the same node.

2. *One updating node per page*
   A further improvement of concurrency is achieved if all locks can be requested at the record level, but a page is being modified in at most one node at a time. Such a scheme is supported by the central locking protocols described in [MN91]. They distinguish between logical record locks requested by individual transactions, and physical page locks requested by the node's buffer managers. While logical locks are held until transaction commit, physical locks are held until the corresponding page is replaced from the buffer (similar to retention locks). To modify a page a physical update lock (U lock) is required; this U lock can be granted to at most one node acting as the page owner. U locks are compatible with physical read locks thereby permitting that a page that is being modified may be concurrently read in other nodes (different records must be accessed since otherwise a lock conflict for the record locks would occur). It is even supported that concurrent transactions of different nodes modify different records of the same page, but the updates are serialized via the physical U locks for the page. The scenario of Figure 7 can be used to illustrate the underlying idea. Assume that a transaction T1 in node P1 has modified a record in page B; P1 therefore holds a physical U lock for B. A second transaction T2 in node P2 can now obtain a logical write lock for a different record of B, but there will be a lock conflict for page B since P2's buffer manager has to request an U lock before T2's update can be performed. This lock conflict is resolved as shown in Figure 7 by revoking P1's U lock and transferring the modified page B (containing the uncommitted update of T1) to P2. P2 now is the new page owner for B.
   Although the scheme of [MN91] supports a high degree of concurrency, it suffers from a very high number of messages for concurrency control. This is not only because the number of record locks is generally much higher than the number of page locks but also because in their scheme the number of messages per lock is higher as for page-level locking. Physical page locks cannot be used here to locally grant logical locks on the respective pages since physical update and read locks are compatible with each other. As a result, the scheme requires separate messages to request logical record and physical page locks from the global lock manager. This causes between 2 and 7 messages per lock! In the best case when an appropriate physical lock is already held, "only" the record lock needs to be requested at the GLM (2 messages). An ownership transfer as in Figure 7b causes 7 instead of 5 messages since 2 extra messages for the record lock are needed. These high message requirements could largely destroy performance gains that may be obtained because of reduced data contention.

3. *Concurrent page updates in different nodes*
   Full record-level locking is achieved if multiple nodes can concurrently modify different records of the same page without serializing the updates at the page level. As indicated in Figure 9, in this case multiple copies of the same page are being modified so that the

modifications need to be merged. The implementation of such a form of record-level locking for database sharing has been described in [Ra89] and [MNS91]. Both proposals are limited to update operations that leave the structure of the page unchanged (e.g., modification of existing records). Page transfers are largely avoided in both approaches by only transferring the modified records between nodes. This results in reduced bandwidth requirements compared to page-level locking.

The proposal in [MNS91] assumes a central locking protocol, while the scheme in [Ra89] is based on primary copy locking with a combined GLA and page ownership allocation. The latter scheme has the advantage that the number of messages per lock is not higher as for page-level locking. A local lock processing is still supported if the respective record belongs to the local partition, otherwise the lock request can be satisfied with two messages.

An even higher degree of concurrency than with record-level locking is supported by special protocols that permit the same record to be concurrently modified by different transactions. Such protocols have been proposed for centralized DBMS and utilize the semantics of special update operations on so-called high-traffic objects like commutativity of increment and decrement operations [Re82, Ga85, ON86]. In [Hä88], it is discussed how the escrow scheme of [ON86] can be extended for database sharing. In [MLS91], an implementation of the IMS Fast Path protocol for high traffic objects [Ga85] is proposed for database sharing.

4.5 Performance studies

Appendix C shows that most performance studies for database sharing either assumed the simpler FORCE approach or even ignored buffer management and coherency control. A NOFORCE scheme for update propagation was only considered in [HR85, Ra88a,b, DY92, Ra93a]. Many studies also made the assumptions of uniform access distribution and random workload allocation. Uniform distribution of database access is an unrealistic assumption since it denies the existence of locality of reference. Random routing represents the worst case with respect to buffer invalidations. It can result in a high degree of replication in the database buffers which reduces overall hit ratios. For update-intensive workloads, the replication additionally leads to many buffer invalidations further lowering hit ratios. All studies assumed page-level concurrency control.

A trace-driven analysis in [Yu87, YCDI87] showed that buffer invalidations lead to an increase in disk reads for FORCE. It was found that this increase grows with both the buffer size and (linearly) with the number of nodes when the workload is distributed at random [Yu87]. For affinity-based routing, significantly improved hit ratios and fewer buffer invalidations were observed compared to a random routing [YCDI87, Ra88a,b, Ra93a]. The trace-driven studies [Ra88a,b, Ra93a] showed that the negative performance impact of buffer invalidations is by far less pronounced in the case of NOFORCE with a direct exchange of modified pages. This is because the page exchange across the communication system is substantially faster than a disk I/O. Furthermore, some schemes like the primary copy approach even avoid separate delays for page requests by providing pages together with lock grant messages.

In [DDY90a,b, DDY91], the impact of buffer invalidations for database sharing is investigated based on an analytic buffer model for LRU page replacement. Only workloads with a single homogeneous transaction type are supported and lock conflicts are analytically modelled. All three papers assume random routing, a FORCE strategy for update propagation and broadcast invalidation to detect buffer invalidations, but consider different concurrency control schemes. In [DDY90a], a central optimistic concurrency control scheme has been chosen for which it was found that restarted transactions experience fewer buffer invalidations than during their first execution. While this study assumed an uniform access distribution, the effect of "skew" in the reference distribution has been considered in [DDY90b]. For this purpose, it was assumed that the database consists of multiple partitions with different access probabilities; within a partition accesses are uniformly distributed. The skewness was found to increase data contention, hit ratios and the number of buffer invalidations. Furthermore, it was predicted that under skewed access the optimistic scheme may outperform a central locking approach provided sufficient CPU capacity is available (due to the fact that roll-backed transactions could be executed with few I/O delays and buffer invalidations).
In [DDY91, Ra93a], it was shown that even for FORCE the negative effects of buffer invalidations can largely be eliminated if a shared non-volatile semiconductor memory is used to write out modified pages. This is because buffer invalidations mainly occur on frequently updated pages that can be cached in the shared memory for a fast access after a local miss. The same effect can be achieved by permanently allocating frequently updated database files to non-volatile semiconductor memeory [Ra93a].

Several studies compared broadcast and on-request invalidation for detecting buffer invalidations [DIY88, Ra88a,b, DIRY89, DY91]. These studies confirmed that on-request invalidation is clearly the preferred method supporting higher throughput since no extra communication is needed to detect buffer invalidations. The communication overhead of broadcast invalidation grows rapidly with both the update probability and the number of nodes. On the other hand, broadcast invalidation allows for better hit ratios than on-request invalidation because of an immediate removal of obsolete pages. However, this has only a minor impact on performance, in general. In [DIY88], it was shown that the performance of a buffer purge approach is mostly even worse than broadcast invalidation.

The analytical study [DY92] investigates several coherency control approaches using either on-request invalidation or retention locks. However, a fair comparison between the schemes is prevented because FORCE was assumed for the on-request invalidation schemes and NOFORCE for the retention lock schemes. For NOFORCE, it was found that a direct page transfer allows better response times than a page exchange across disk. The retention lock schemes suffered from a high CPU overhead for page transfers in the case of high update probability and low node-specific locality of reference.

In [YD91] a performance comparison between database partitioning systems with "function request shipping"[11] and database sharing with and without a shared intermediate memory has been presented. The database sharing configurations were based on FORCE, central locking and on-request invalidation, while NOFORCE was assumed for database partitioning. Database sharing with a non-volatile shared intermediate memory was found to provide the best throughput and response time results, in particular if the database and workloads cannot effectively be partitioned for "shared nothing" and in cases with workload fluctuations ("load surges") causing unbalanced CPU utilization for database partitioning. Database sharing without such a shared memory may also support better throughput than database partitioning, but response times were found to be worse due to the FORCE assumption and because of lower hit ratios (buffer invalidations, replicated caching of pages in multiple nodes).

11. Function request shipping is used in IBM's TP monitor CICS to distribute database operations (DL/1 calls). A main restriction is that every database operation must completely be processed in one node and that files are the smallest units for database allocation. Since DL/1 is a record-oriented (non-relational) database language with few database accesses per operation, the ratio between communication overhead for a remote operation and useful work is typically very unfavourable.

5. Related Concurrency and Coherency Control Problems

As mentioned in Section 2, workstation/server DBMS, network file systems, and distributed shared memory systems face similar coherency problems than database sharing. They also support replicated main memory caching of pages (or other data granules) in multiple computers so that coherency control becomes necessary. Some systems also support concurrency control. In this section, we briefly summarize the main differences to database sharing and the major solutions that are pursued in the respective areas.

5.1 Workstation/server DBMS

In workstation/server DBMS, database functionality is partitioned between a server DBMS and a workstation DBMS [DFMV90]. Such an architecture is primarily used by object-oriented DBMS that want to exploit the graphical interface, processing power and memory capacity of workstations for complex scientific and engineering database applications. The server DBMS manages external storage devices and performs global services like logging or concurrency control. While workstations may be diskless, database pages (or objects) are cached in main memory by the workstation DBMS. This can reduce the communication frequency with the server, but also gives rise to coherency problems.

Despite the different architecture, many of the database sharing techniques for concurrency and coherency control can be employed for workstation/server DBMS as well. In fact, most of the algorithms that are used in existing workstation/server DBMS (e.g., Orion [KGBW90], ObjectStore [LLOW91]) or have been studied in recent publications (e.g., [WN90, CFLS91, WR91, FC92, FCL92]) are variants of the schemes that were developed for database sharing. These systems and studies typically assume a single server acting as both a global lock manager and a central page owner. Furthermore, a FORCE-like update strategy between workstation and server DBMS is mostly assumed where all modifications of a transaction (as well as log data) are propagated to the server at transaction commit. For such a configuration, all database sharing techniques to reduce the number of global lock requests (Section 3.1.2) and for detection or avoidance of buffer invalidations (Section 4.1) can directly be used.

Orion employs an on-request invalidation technique to detect buffer invalidations [KGBW90], while ObjectStore avoids buffer invalidations by using retention locks (named "callback locks") [LLOW91]. The use of retention locks for concurrency and coherency control has also been studied in [WR90, FC92, FCL92]. In [CFLS91, FC92], on-request invalidation (in combination with central locking) as well as several optimistic concurrency control schemes were considered. In the optimistic schemes, it was assumed that the server keeps track at which workstations pages are cached. During the validation of a transaction, these "copy sites" are either provided with the new page versions or their page copies get invalidated (similar to selective broadcast invalidation). In [FCL92], a direct exchange of pages between workstations has been considered to speed up update propagation. However, the scheme depends on the server information about copy sites so that communication with the server is still necessary after a local buffer miss. Only when the server has no cached version of the requested page, the page request is forwarded to one of the copy sites rather than reading the page from disk.

A major limitation of the workstation/server studies and existing implementations is the restriction to a single server node. To avoid such a single point of failure and a potential performance bottleneck, a distributed server system is highly desirable. A distributed server may be based on either a database partitioning or database sharing architecture. The use of the database sharing approach seems advantageous because similar concurrency and coherency control schemes could be used in the server system as between workstation and server DBMS. Another limitation is the use of the conventional transaction concept which is not adequate for advanced database applications like CAD or software engineering [BK91]. Workstation/server cooperation to support engineering database applications is discussed in [HHMM88].

Workstation caching of database objects can also be useful for information retrieval systems. Such an approach has been proposed in [ABG90], where so-called "quasi-copies" may be cached at a user's workstation in read-only mode. By supporting user-defined consistency constraints (e.g., periodic refreshing of cached data), the overhead for update propagation can be kept small thereby improving scalability.

5.2 Network file systems

Network file systems support shared file access for multiple clients in a collection of loosely coupled nodes interconnected by a local area network. Similar to workstation/server DBMS, clients typically execute in workstations while one or multiple server provide common file services. To reduce the communication frequency with the server, caching of files is supported in the client nodes. Some systems (e.g., Andrew [Ho88]) only allow caching of entire files, while most systems use page caching (e.g., Sun's Network File System and Sprite [LS90]).

A major difference to database sharing is that network file systems typically do not support transactions. As a result, transaction-based concurrency control and recovery functions are not provided. Furthermore, coherency requirements are related to file operations (open/close, read/write) rather than to transactions. A desirable consistency approach is to guarantee that every read operation on a file sees the effects of all previous writes. This consistency requirement is supported by Sprite [NWO88]. In their implementation, however, concurrent caching of a file in different nodes is only supported for read access.

Despite these differences, similar coherency control schemes than for database sharing can be used. An overview of coherency control in network file systems is provided in [LS90]. To detect invalidated pages or files, two major policies are followed. In a client-initiated approach, before accessing cached data the client checks the validity of a cached file or page at the server. This scheme corresponds to on-request invalidation for database sharing but requires extra messages for the validity checks. In a server-initiated approach, the file server keeps track of the clients' cache contents. After the server is informed of a file modification, it explicitly notifies all clients that may hold invalidated copies of the respective file. Such an approach corresponds to the selective broadcast invalidation scheme for database sharing. The Andrew file system uses a server-initiated coherency scheme in combination with a callback mechanism that is similar to the use of retention locks [Ho88].

For update propagation, network file systems either use a write-through, write-on-close or delayed-write policy [LS90]. Write-through requires that every modification at a client is directly sent to the server, while write-on-close sends modified data to the server when the file is closed. These policies roughly correspond to a FORCE scheme with respect to write operations (write-through) or file sessions (write-on-close). A delayed-write scheme propagates updates to the sever after a predetermined time threshold or when the data is to be replaced from the cache. This approach corresponds to NOFORCE and allows for reduced communication overhead. For write-on-close and delayed-write, a client failure results in a loss of all modifications that were not yet propagated to the server at crash time.

5.3 Distributed shared memory

Distributed shared memory (DSM) systems provide applications in a distributed system with a shared memory abstraction [NL91]. The main goal is to simplify development of distributed applications compared to the conventional approach based on remote procedure calls or explicit message passing operations for inter-process communication. The operating system implements the shared memory paradigm on top of a loosely coupled system with physically distributed memory, so that a logical memory reference may require requesting the respective data (page) from a remote node. Caching is essential to reduce the number of remote memory accesses, but also necessitates coherency control.

DSM systems typically do not support transactions resulting in different coherency requirements than for database sharing. Most DSM systems enforce strict memory consistency where each read operation returns the most recently written value. To improve performance, some DSM systems support weaker forms of memory coherency by utilizing application semantics [NL91]. However, these approaches require the application programmer to understand and obey the respective consistency model to obtain correct programs. In contrast, database sharing systems provide full distribution transparency to application programs. In DSM systems, all memory pages are exchanged across the communication system since there are no shared disks.

DSM is a very active area of research and numerous prototype systems have been built [He90, MR91, NL91]. As in database sharing systems, (horizontal) page migration is mostly based on an ownership approach with either a central page owner or fixed or dynamically distributed page ownerships [LH89]. To avoid access to invalidated pages, most DSM sys-

tems adopt a "write-invalidate" approach where all page copies are explicitly invalidated after a modification. This approach is analogous to the broadcast invalidation or selective invalidation scheme of database sharing systems (in the DSM implementations, the owner of a page usually keeps track of the copy nodes).
In the Clouds distributed operating system, invalidation messages are avoided by combining coherency control with locking [RAK89]. In their implementation, the page owner is also responsible for synchronizing memory accesses. This allows combining page transfers with lock request and release messages similar to the database sharing schemes with corresponding GLA and page ownership assignment (Section 4.2.2). However, in Clouds the copy of a page must be eliminated from a node's memory when the lock is released. Hence, this scheme corresponds to a buffer purge approach for avoiding buffer invalidations significantly limiting the cache residence time of pages so that locality of reference cannot fully be exploited. Furthermore, lock and unlock operations have to be specified by the application programmer, while for database sharing concurrency and coherency control are automatically performed by the DBMS.

In [BHT90], concurrency and coherency control schemes for transaction-based DSM systems were studied. In such an environment, all database sharing schemes (for NOFORCE) can directly be used. They studied a distributed locking protocol and two optimistic schemes. The locking scheme is based on dynamic page ownerships and a dynamic GLA assignment that changes together with the page ownership. The locating directory is distributed among all nodes; read and write authorizations are supported. Both optimistic schemes use a central node for validation, but differ with respect to update propagation (central page owner vs. distributed fixed page ownership).

6. Summary

Database sharing is an attractive approach for distributed transaction and database processing since there is no need to find a physical database partitioning as in so-called shared nothing systems.This facilitates a migration from centralized to distributed transaction processing and provides advantages with respect to load balancing. The performance of database sharing systems critically depends on the algorithms used for concurrency and coherency control because these functions largely determine the amount of inter-node communication. New technical solutions for database sharing are further required in the areas of workload management (in particular for transaction routing), logging and recovery.

We have presented a classification and survey of concurrency control and coherency control methods for database sharing. With respect to concurrency control we considered locking and optimistic schemes under central or distributed control. For the locking protocols, we have outlined four basic approaches as well as several extensions that can be employed to reduce the number of global lock requests. The coherency control problem was decomposed into two subproblems: detection or avoidance of buffer invalidations and update propagation. The major solutions for both subproblems have been described and it was illustrated how they may be combined within a complete coherency control protocol. Furthermore, we have indicated how the various concurrency control and coherency control schemes can be combined with each other and which combinations promise the best performance. In addition, methods to reduce data contention were discussed like multi-version concurrency control and record-level locking. We also provided an overview of the main findings of database sharing performance studies and discussed concurrency and coherency control in related system architectures.

Results from trace-driven simulations indicate that even optimized optimistic concurrency control schemes appear inadequate for most real-life workloads since they often cause an excessive number of transaction rollbacks. Data contention can also be a serious performance problem for locking protocols if only page-level locking is supported (hot spot pages); at least a limited form of record-level locking seems therefore indispensable. The concepts to reduce the number of global lock requests (hierarchical locking, use of a local GLA, sole interest, read optimization) have been shown to be effective to varying degrees, albeit they largely depend on workload characteristics and the strategy used for workload allocation. In general, affinity-based transaction routing is essential for good performance of a database sharing system not only to reduce the number of global lock requests, but also to reduce the number of buffer invalidations and page transfers. Several performance studies have confirmed that efficient coherency control should be based on either on-request invalidation or the use of retention locks rather than on broadcast invalidation or even a buffer purge scheme. To support high performance, update propagation should use a NOFORCE strategy for updating the permanent database on disk as well as a direct exchange of modified objects (pages, records) over the communication system.

These observations primarily hold for database sharing systems using conventional hardware for communication and data storage. Special hardware support could make it affordable to globally decide upon all lock requests thereby reducing the dependencies on workload characteristics and affinity-based routing (e.g., global locking could be performed by a dedicated lock engine or via a global lock table in shared semiconductor memory). Similarly, a FORCE scheme for update propagation may be acceptable if a shared non-volatile semiconductor memory is available to speed up the write I/Os. Such a store can also alleviate the negative performance impact of buffer invalidations since it could hold the current version of frequently modified (invalidated) pages and provide them to all nodes. On the other hand, such special hardware devices typically incur a high cost and may limit the number of nodes that can be supported. Furthermore, new availability problems arise that need to be solved.

Several important problems remain to be addressed in future research or system developments for database sharing. In particular, more work is needed on dynamic load balancing schemes that are able to fully utilize the potential offered by the database sharing architecture. Furthermore, parallel query processing strategies tailored to database sharing should be investigated in order to support short response times for complex and data-intensive database operations.

Appendices

Appendix A: Concurrency / coherency control in existing database sharing implementations

system	***concurrency control***	***coherency control***	***references***
IMS Data Sharing	pass-the-buck	broadcast invalidation FORCE	[SUW82, Yu87]
Amoeba prototype	central lock manager	broadcast invalidation FORCE	[Tr83, Sh85]
DEC	dynamic lock authority	on-request invalidation FORCE+	[KLS86, RSW89, Jo91]
Computer Console*	central lock manager	FORCE	[WIH83]
Oracle*	central lock manager exchange of modified pages over disk	NOFORCE	[Or90]

*Concurrency/coherency control protocols not completely described in available documentation

+ DEC’s relational DBMS Rdb supports NOFORCE since its version 4.1.

Appendix B: Selected papers on concurrency / coherency control for database sharing

paper	***concurrency control***	***coherency control***	***remarks***
HR85	pass-the-buck (extended)	retention locks NOFORCE exchange of modified pages over disk dynamic page ownership	
Ra86	fixed lock authority (primary copy locking)	on-request invalidation NOFORCE direct exchange of modifications fixed or dynamic page ownership	
Ra87a,b	distributed OCC	asynchronous broadcast invalidation NOFORCE direct exchange of modifications dynamic page ownership	
DIRY89	central lock engine	on-request invalidation FORCE	special-purpose lock processor
MN91	central lock manager	on-request invalidation NOFORCE different page transfer schemes dynamic page ownership	record-level locking (see 4.4.2)
MN92a	central lock manager	retention locks NOFORCE "medium" page transfer scheme dynamic page ownership	(see footnote 6)

Appendix C: Database sharing performance studies

The table below summarizes which concurrency / coherency control protocols have been analysed in database sharing performance evaluations. Under "methodology" we indicate whether trace-driven simulations, simulations with synthetic workloads, or analytical modelling and which type of queuing model (open or closed) have been employed. In closed models, typically throughput has been used as the primary performance measure, while open models mostly concentrated on response times. Under "focus" we indicate the major aspects that have been analysed in the respective study.
Entries are ordered by year and alphabetically within a year.

paper	concurrency control	coherency control	methodology / focus
HR85	pass-the-buck (extended)	retention locks, NOFORCE exchange of modified pages over disk dynamic page ownership	trace-driven sim., closed model protocol analysis; influence of different (6) workloads, token delay, number of nodes (1-2)
CDY86	central lock engine	broadcast inv., FORCE	simulation/analytical; open model focus on "shared nothing"; comparison with database sharing
YCDI87	pass-the-buck	broadcast inv., FORCE	simulation/analytical; open model protocol analysis; affinity-based routing; buffer invalidation study
Yu87	pass-the-buck central lock engine	broadcast inv., FORCE - " -	simulation/analytical; open model protocol analysis; affinity-based routing; buffer invalidation study
Bh88	central lock manager primary copy locking disk controller locking	ignored (buffer purge) - " - - " -	synth. simulation; closed model protocol comparison; influence of message bundling and intra-transaction parallelism; comparison with "shared memory"/"shared nothing"
DIY88	distributed fixed GLA - " - - " -	broadcast inv., FORCE on-request inv., FORCE buffer purge, FORCE	analytical; open model cost-effectiveness study; influence of number of nodes (1-30), MIPS per node (1-100) and coherency control strategy
IK88	distributed fixed GLA distributed dynamic GLA	ignored - " -	analytical; closed model # messages per lock
Ra88a, b	primary copy locking central lock manager central optimistic distributed optimistic	on-request inv., NOFORCE direct exchange of modif. fixed page ownership broadcast inv., NOFORCE direct exchange of modif. dynamic page ownership - " - - " -	trace-driven sim., closed model protocol comparison; influence of different workloads, number of nodes (1-4), routing strategy, hot spot objects, comm. overhead

Appendix C (continued): Database sharing performance studies

paper	*concurrency control*	*coherency control*	*methodology / focus*
DIRY89	central lock engine	broadcast inv., FORCE	analytical; open model
	- " -	on-request inv., FORCE	coherency control; use of volatile shared intermediate memory
DDY90a	central optimistic	broadcast inv., FORCE	analytical; open model analysis of hit ratios and buffer invalidations
DDY90b	central optimistic	broadcast inv., FORCE	synth. simulation; closed model analysis of hit ratios and buffer invalidations; influence of skew and concurrency control scheme
	central lock manager	- " -	
DY91	central lock manager	broadcast inv., FORCE	analytical; open model
	- " -	selective inv., FORCE	comparison of coherency control strategies; hit ratios
	- " -	on-request inv., FORCE	
DDY91	central lock manager	broadcast inv., FORCE	analytical; open model use of non-vol. shared intermediate memory; buffer invalidations
YD91	central lock manager	on-request inv., FORCE	analytical; open model comparison with "shared nothing"; use of non-vol. intermed. memory; influence of database partitiona-bility and load surges
DY92	central lock manager	on-request inv., FORCE	analytical; open model
	- " -	retention locks, NOFORCE page exchange across disk dynamic page ownership	
	- " -	retention locks, NOFORCE direct exchange of modif. dynamic page ownership	comparison of coherency control protocols
Ra93a	primary copy locking	on-request inv., FORCE + NOFORCE direct exchange of modif. fixed page ownership	trace-driven and synt. sim., open model
	global lock table in shared global memory	on-request inv. FORCE + NOFORCE direct exchange of modif. dynamic page ownership	close vs. loose coupling; FORCE vs. NOFORCE impact of workload allocation and buffer invalidations

References

ABG90	Alonso, R., Barbara, D., Garcia-Molina, H.: Data caching issues in an information retrieval system. *ACM Trans. Database Systems 15*, 3, 359-384, 1990
AIM86	AIM/SRCF functions and facilities. Facom OS Techn. Manual 78SP4900E, Fujitsu Limited, 1986
BDS79	Behman, S.B., Denatale, T.A., Shomler, R.W.: Limited lock facility in a DASD control unit. Technical report TR 02.859, IBM General Products Division, San Jose, 1979
BHT90	Bellew, M., Hsu, M., Tam, V.: Update propagation in distributed memory hierarchies. *Proc. 6th Int. Conf. on Data Engineering*, IEEE Computer Society Press, 521-528, 1990
BHG87	Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison Wesley, 1987.
Bh88	Bhide, A.: An analysis of three transaction processing architectures. *Proc. 14th Int. Conf. on Very Large Data Bases,* Long Beach, CA, 339-350, 1988
BK91	Barghouti, N.S., Kaiser, G.E.: Concurrency control in advanced database applications. *ACM Comput. Surv. 23*, 3, 269-317, 1991.
Bo81	Borr, A.: Transaction monitoring in Encompass: A non shared-memory multi-processor approach. *Proc. 7th Int. Conf. on Very Large Data Bases*, 155-165, 1981
Bo90	Boral, H. et al.: Prototyping Bubba: A highly parallel database system. *IEEE Trans. Knowledge and Data Engineering 2,* 1, 4-24, 1990
BT90	Burkes, D.L. Treiber, R.K.: Design approaches for real-time transaction processing remote site recovery. *Proc. IEEE Spring CompCon*, 568-572, 1990
CDY86	Cornell, D.W., Dias, D.M., Yu, P.S.: On multisystem coupling through function request shipping. *IEEE Trans. Soft. Eng. 12*, 10, 1006-1017, 1986
CFLS91	Carey, M.J., Franklin, M.J., Livny, M., Shekita, E.J.: Data caching tradeoffs in client-server DBMS architectures. *Proc. ACM SIGMOD Conf.*, Boulder, 357-366, 1991.
CG85	Chan, A., Gray, R.: Implementing distributed read-only transactions. *IEEE Trans. Soft. Eng. 11*, 2, 205-212, 1985.
Ch82	Chan, A. et al.: The implementation of an integrated concurrency control and recovery sceme. *Proc. ACM SIGMOD Conf.,* 184-191, 1982.
CLSW84	Cheng, J.M., Loosley, C.R., Shibamiya, A., Worthington, P.S.: IBM Database 2 performance: design, implementation and tuning. *IBM Systems Journal 23*, 2, 189-210, 1984.
CM86	Carey, M.J., Muhanna, W.A.: The performance of multiversion concurrency control algorithms. *ACM Trans. Comp. Systems 4*, 4, 338-378, 1986.
CM88	Chang, A., Mergen, M.F.: 801 storage: architecture and programming. *ACM Trans. Comp. Systems 6*, 1, 28-50, 1988.
CP84	Ceri, S., Pelagatti, G.: Distributed databases. Principles and systems. Mc Graw-Hill, 1984.
DDY90a	Dan, A., Dias, D.M., Yu, P.S.: Database buffer model for the data sharing environment. *Proc. 6th Int. Conf. on Data Engineering*, IEEE Computer Society Press, 538-544, 1990.
DDY90b	Dan, A., Dias, D.M., Yu, P.S.: The effect of skewed data access on buffer hits and data contention in a data sharing environment. *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, 419-431, 1990.
DDY91	Dan, A., Dias, D.M., Yu, P.S.: Analytical modelling of a hierarchical buffer for the data sharing environment. *Proc. ACM SIGMETRICS Conf.*, 156-167, 1991
De90	DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H., Rasmussen, R.: The Gamma database machine project. *IEEE Trans. Knowledge and Data Engineering 2* ,1, 44-62, 1990
DFMV90	DeWitt, D.J., Futtersack, P., Maier, D., Velez, F.: A study of three alternative workstation-server architectures for object oriented database systems. *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, 107-121, 1990.
DG92	DeWitt, D.J., Gray, J. : Parallel database systems: the future of high performance database systems. *CACM 35* , 6, 85-98, 1992.
DIRY89	Dias, D.M., Iyer, B.R., Robinson, J.T., Yu, P.S.: Integrated concurrency-coherency controls for multisystem data sharing. *IEEE Trans. Soft. Eng. 15*, 4, 437-448, 1989.

DIY88 Dias, D.M., Iyer, B.R., Yu, P.S.: Tradeoffs between coupling small and large proces for transaction processing. *IEEE Trans. Comp. 37*, 3, 310-320, 1988.

DY91 Dan, A., Yu, P.S.: Performance comparisons of buffer coherency policies. *Proc. 11tl Conf. on Distributed Computing Systems*, Arlington, TX, IEEE Computer Society Pı 208-217, 1991.

DY92 Dan, A., Yu, P.S.: Performance analysis of coherency control policies through lock re tion. *Proc. ACM SIGMOD Conf.*, San Diego, CA, 114-123, 1992.

EGLT76 Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and p icate locks in a database system. *CACM 19*, 11, 624-633, 1976.

EH84 Effelsberg, W., Härder, T.: Principles of database buffer management. *ACM Trans. tabase Systems 9*, 4, 560-595, 1984.

FC92 Franklin, M.J., Carey, M.J.: Client-server caching revisited. *Proc. Int. Workshop on tributed Object Management*, Edmonton,1992.

FCL92 Franklin, M.J., Carey, M.J., Livny, M.: Global memory management in client-server D architectures. *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver,596-609, 1

FRT90 Franaszek, P.A., Robinson, J.T., Thomasian, A.: Access invariance and its use in contention environments. *Proc. 6th Data Engineering Conf.*, IEEE Computer So Press, 47-55, 1990.

Ga85 Gawlick, D.: Processing 'hot spots' in high performance systems. *Proc. IEEE Sp CompCon*, 249-251, 1985.

GA87 Garcia-Molina, H., Abbott, R.K.: Reliable distributed database management. *Proceed of the IEEE 75*, 5, 601-620, 1987

GGHS85 Gray, J., Good, B., Gawlick, D., Homan, P., Sammer, H.: One thousand transactions second. *Proc. IEEE Spring CompCon*, 96-101, 1985

GLPT76 Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.: Granularity of locks and degrees of sistency in a shared data base. *Proc. IFIP Working Conf. on Modelling in Data Base I agement Systems*, North-Holland, 365-394, 1976

Gr78 Gray, J. N.: Notes on data base operating systems. In: 'Operating Systems - An vanced Course', Lecture Notes in Computer Science 60, Springer-Verlag, 393-481, 1

Gr81 Gray, J. N.: The transaction concept: virtues and limitations. *Proc. 7th Int. Conf. on Large Data Bases*, Cannes, France, 144-154, 1981.

Gr91 Gray, J. (ed.): The benchmark handbook for database and transaction processing tems. Morgan Kaufmann Publishers, 1991.

Gr92 Grossman, C.P.: Role of the DASD storage control in an enterprise systems connectior vironment. *IBM Systems Journal 31*, 1, 123-146, 1992

GR93 Gray, J., Reuter, A.: Transaction processing - concepts and techniques. Morgan Kaufm 1993

GS91 Gray, J., Siewiorek, D.P.: High-availability computer systems. *IEEE Computer,* 3 Sept. 1991

Gu92 Guterl, F.V.: Twin mainframes power Lufthansa's reservations. *Datamation*, 95-96, 1, 1992.

Ha90 Hastings, A.B.: Distributed lock management in a transaction processing environm *Proc. 9th Symposium on Reliable Distributed Systems*, Huntsville, IEEE Computer S ety Press, 22-31, 1990.

Hä84 Härder, T.: Observations on optimistic concurrency control. *Information Systems* 111-120, 1984.

Hä88 Härder, T.: Handling hot spot data in DB-sharing systems. *Information Systems 1* 155-166, 1988.

He90 Hellwagner, H.: A survey of virtually shared memory schemes. Research report T I9056, Technical Univ. of Munich, 1990.

HG89 Herman, G., Gopal, G.: The case for orderly sharing. *Proc. 2nd Int. Workshop on Performance Transaction Systems*, Asilomar, CA. Lecture Notes in Computer Scie Vol. 359, Springer-Verlag, Berlin, 148-174, 1989.

HHMM88 Härder, T., Hübel, C., Meyer-Wegener, K., Mitschang, K.: Processing and transa concepts for cooperation of engineering workstations and a database server. *Da Knowledge Engineering* 3, 87-107, 1988.

Ho88 Howard, J.H. et al.: Scale and performance in a distributed file system. *ACM Trans. Comp. Systems 6*, 1, 51-81, 1988.

HP87 Härder, T., Petry, E.: Evaluation of a multiple version scheme for concurrency control. *Information Systems 1*2, 1, 83-98, 1987.

HR83 Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4, 287-317, 1983.

HR85 Härder, T., Rahm, E.: Quantitative analysis of a synchronization protocol for DB-sharing. *Proc. 3. GI/NTG Conf. on Measurement, Modelling and Evaluation of Computer Systems*, Informatik-Fachberichte 110, Springer-Verlag , 186-201, 1985 (in German).

IK88 Iyer, B.R. Krishna, C.M.: Tradeoffs between static and dynamic lock name space partitioning. IBM Research Report RJ 6566, San Jose, CA, 1988

IYD87 Iyer, B.R., Yu, P.S. , Donatiello, L.: Analysis of fault-tolerant multiprocessor architectures for lock engine design. *Computer Systems Science and Engineering 2*, 2, 59-75, 1987.

Jo91 Joshi, A.M.: Adaptive locking strategies in a multi-node data sharing environment. *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, 181-191, 1991

JR89 Joshi, A. M. Rodwell, K.E.: A relational database management system for production applications. *Digital Technical Journal*, No. 8, 99-109, Feb. 1989.

KGBW90 Kim, W., Garza, J.F., Ballou, N. Woelk, D.: Architecture of the Orion next-generation database system. *IEEE Trans. Knowledge and Data Engineering 2* ,1, 109-124, 1990.

KHGP91 King, R.P., Halim, N., Garcia-Molina, H., Polyzois, C.A.: Management of a remote backup copy for disaster recovery. *ACM Trans. Database Systems 16*, 2, 338-368, 1991.

Ki84 Kim, W.: Highly available systems for database applications. *ACM Comput. Surv. 16*, 1, 71-98, 1984.

KLS86 Kronenberg, N.P., Levy, H.M., Strecker, W.D.: VAX clusters: a closely coupled distributed system. *ACM Trans. Comp. Systems 4*, 2, 130-146, 1986.

Kn87 Knapp, E.: Deadlock detection in distributed databases. *ACM Comput. Surv. 19*, 4, 303-328, 1987.

KR81 Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. *ACM Trans. Database Systems 6*, 2, 213-226, 1981.

LH89 Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. *ACM Trans. Comp. Systems 7*, 4, 321-359, 1989.

LLOW91 Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore database system. *CACM 34*, 10, 50-63, 1991.

Lo90 Lomet, D.: Recovery for shared disk systems using multiple redo logs. Technical Report CRL 90/4, DEC Cambridge Research Lab., Cambridge, MA, 1990.

LS90 Levy, E., Silberschatz, A.: Distributed file systems: concepts and examples. *ACM Comput. Surv. 22*, 4, 321-374, 1990.

MN91 Mohan, C., Narang, I.: Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, 193-207, 1991.

MN92a Mohan, C., Narang, I.: Efficient locking and caching of data in the multisystem shared disks transaction environment. *Proc. 3rd Int. Conf. on Extending Database Technology*, Vienna, Lecture Notes in Computer Science 580, Springer-Verlag, 453-468, 1992.

MN92b Mohan, C., Narang, I.: Data base recovery in shared disks and client-server architectures. *Proc. 12th Int. Conf. on Distributed Computing Systems*, Yokohama, IEEE Computer Society Press, 1992.

MNS91 Mohan, C., Narang, I., Silen, S.: Solutions to hot spot problems in a shared disks transaction environment. *Proc. 4th Int. Workshop on High Performance Transaction Systems*,, Asilomar, CA, 1991.

Mo92 Mohan, C.: Less optimism about optimistic concurrency control. *Proc. 2nd Workshop on Research Issues on Data Engineering (RIDE-2)*, Tempe, AZ, IEEE Computer Society Press, 199-204, 1992.

MR91 Mohindra, A., Ramachran, U.: A survey of distributed shared memory in loosely-coupled systems. Technical report GIT-CC-91/01, Georgia Institute of Technology, Atlanta, 1991.

Ne86 Neches, P.M.: The anatomy of a database computer - revisited. *Proc. Spring CompCon Conf.*, IEEE Computer Society Press, 374-377, 1986.

NL90 Nitzberg, B., Lo, V.: Distributed shared memory: a survey of issues and algorithms. *I Computer* , August1990.

NWO88 Nelson, M.N., Welch, B.B., Ousterhout, J.K.: Caching in the Sprite network file sys *ACM Trans. Comp. Systems 6*, 1,134-154, 1988.

ON86 O'Neil, P.E.: The Escrow transactional method. *ACM Trans. Database Systems 1* 405-430, 1986.

Or90 Oracle for massively parallel systems - technology overview. Oracle Corporation, number 50577-0490, 1990.

Or91 TPC Benchmark B - Full disclosure report for the nCUBE 2 scalar supercomputer m nCDB-1000 using Oracle V6.2. Oracle Corporation, part number 3000097-0391, 19

ÖV91 Özsu, M.T., Valduriez, P.: Principles of distributed database systems. Prentice-Hall,

Pi90 Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P.: Parallelism in relational base systems: architectural issues and design approaches. *Proc. 2nd Int. Symposiu Databases in Parallel and Distributed Systems*, Dublin, IEEE Computer Society P 1990.

Ra86 Rahm, E.: Primary copy synchronization for DB-sharing. *Information Systems 11*, 4, 286, 1986.

Ra87a Rahm, E.: Integrated solutions to concurrency control and buffer invalidation in data sharing systems. *Proc. 2nd Int. Conf. on Computers and Applications*, Peking, IEEE C puter Society Press, 410-417, 1987

Ra87b Rahm, E.: Design of optimistic methods for concurrency control in database sharing tems. *Proc. 7th Int. Conf. on Distributed Computing Systems*, West Berlin, IEEE Com er Society Press, 154-161, 1987.

Ra88a Rahm, E.: Concurrency control in multi-computer database systems (in German). I matik-Fachberichte 186, Springer-Verlag, 1988.

Ra88b Rahm, E.: Empirical performance evaluation of concurrency and coherency control tocols for database sharing. IBM Research Report RC 14325, IBM T.J. Watson Rese Center, Yorktown Heights, NY, 1988. An extended version of this paper will appe *ACM Trans. Database Systems 18*, 2, 1993.

Ra89 Rahm, E.: Recovery concepts for data sharing systems. Technical report 14/89, Com er Science Dept., Univ. Kaiserslautern. A shorter version of this paper appeared in *F of the 21st Int. Symposium on Fault-Tolerant Computing*, Montreal, IEEE Computer ciety Press, 368-375.

Ra91 Rahm, E.: Use of global extended memory for distributed transaction processing. *F 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, CA,1991

Ra92 Rahm, E.: A framework for workload allocation in distributed transaction processing tems. *Journal of Systems and Software 18*, 3, 171-190, 1992.

Ra93a Rahm, E.: Evaluation of closely coupled systems for high performance database cessing. *Proc. 13th Int. Conf. on Distributed Computing Systems*, Pittsburgh, IEEE C puter Society Press,1993.

Ra93b Rahm, E.: Parallel query processing in shared disk database systems. Techn. Re Univ. of Kaiserslautern, 1993

RAK89 Ramachran, U., Ahamad, M., Khalidi, M.Y.A.: Coherence of distributed shared mem unifying synchronization and data transfer. *Proc. Int. Conf. on Parallel Processing,* II, 160-169, 1989.

Re82 Reuter, A.: Concurrency on high-traffic data elements. *Proc. ACM SIGACT-SIGI Symp. on Principles of Database Systems*, 83-92, 1982.

Re86 Reuter, A.: Load control and load balancing in a shared database management sys *Proc. 2nd Int. Conf. on Data Engineering*, Los Angeles, IEEE Computer Society P 188-197, 1986.

Ro85 Robinson, J.T.: A fast general-purpose hardware synchronization mechanism. *Proc. SIGMOD Conf.*, 122-130, 1985

RS84 Reuter, A., Shoens, K.: Synchronization in a data sharing environment. Technical Re IBM San Jose Research Lab., 1984

RSW89 Rengarajan, T.K., Spiro, P.M., Wright, W.A.: High availability mechanisms of VAX D software. *Digital Technical Journal*, No. 8, 88-98, Feb. 1989.

RT87	Ryu, I.K., Thomasian, A.: Performance analysis of centralized databases with optimistic concurrency control. *Performance Evaluation 7*, 3, 195-211, 1987
Sc87	Scrutchin Jr., T.W.: TPF: performance, capacity, availabilty. *Proc. Spring CompCon*, IEEE Computer Society Press,158-160, 1987.
Se84	Sekino, A. , Moritani, K., Masai, T., Tasaki, T., Goto, K.: The DCS - a new approach to multisystem data sharing. *Proc. National Computer Conf.*, Las Vegas, 59-68, 1984.
Sh86	Shoens, K.: Data sharing vs. partitioning for capacity and availability. *IEEE Database Engineering 9*, 1, 10-16, 1986.
Sh85	Shoens, K., Narang, I., Obermarck, R., Palmer, J., Silen, S., Traiger, I., Treiber, K.: The Amoeba project. *Proc. Spring CompCon*, IEEE Computer Society Press, 102-105, 1985.
Sm82	Smith, A.J.: Cache memories. *ACM Comput. Surv. 14*, 3, 473-530, 1982.
ST87	Snaman Jr., W.E., Thiel, D.W.: The VAX/VMS distributed lock manager. *Digital Technical Journal*, No. 5, 29-44, Sep. 1987
St79	Stonebraker, M.: Concurrency control and consistency of multiple copies in distributed Ingres. *IEEE Trans. Soft. Eng. 5*, 3, 188-194, 1979
St84	Stonebraker, M.: Virtual memory transaction management. *ACM Operating Systems Review 18,* 2, 8-16, 1984.
St86	Stonebraker, M.: The case for shared nothing. *IEEE Database Engineering 9*,1, 4-9, 1986
St90	Stenström, P.: A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 12-24, June 1990
SUW82	Strickland, J., Uhrowczik, P., Watts, V.: IMS/VS: an evolving system. *IBM Systems Journal 21*, 4, 490-510, 1982
SZ90	Stumm, M., Zhou, S.: Algorithms implementing distributed shared memory. *IEEE Computer*, 54-64, May 1990
Ta89	The Tandem Database Group: NonStop SQL, a distributed, high-performance, high-availability implementation of SQL. Lecture Notes in Computer Science 359, Springer-Verlag, 60-104, 1989 (*Proc. 2nd Int. Workshop on High Performance Transaction Systems, 1987*).
TPF88	Transaction Processing Facility, Version 2 (TPF2). General Information Manual, Release 4.0, IBM Order No. GH20-7450, 1988
TR90	Thomasian, A., Rahm, E.: A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking. *Proc. 10th Int. Conf. on Distributed Computing Systems*, Paris, IEEE Computer Society Press, 294-301, 1990.
Tr83	Traiger, I.: Trends in systems aspects of database management.*Proc. British Computer Society 2nd Int. Conf. on Databases*, Cambridge, England, 1-20, 1983.
TW91	Trew, A., Wilson, G. (eds.): Past, present, parallel. Springer-Verlag, 1991.
We87	Weihl, W.E.: Distributed version management for read-only actions. *IEEE Trans. Soft. Eng. 13*, 1, 55-64, 1987.
WIH83	West, J.C., Isman, M.A., Hannaford, S.G.: PERPOS fault-tolerant transaction processing. *Proc. 3rd Symposium on Reliability in Distributed Software and Database Systems*, IEEE Computer Society Press,189-194, 1983.
WN90	Wilkinson, K., Neimat, M.: Maintaining consistency of client-cached data. *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane,122-133, 1990.
WR91	Wang, Y., Rowe, L.A.: Cache consistency and concurrency control in a client/server DBMS architecture. *Proc. ACM SIGMOD Conf.*, Boulder, 367-376, 1991.
YCDI87	Yu, P.S., Cornell, D.W., Dias, D.M., Iyer, B.R.: Analysis of affinity based routing in multisystem data sharing. *Performance Evaluation 7*, 2, 87-109, 1987.
YD91	Yu, P.S., Dan, A.: Comparison on the impact of coupling architectures to the performance of transaction processing systems. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, CA, 1991.
Yu87	Yu , P.S., Dias, D.M., Robinson, J.T., Iyer, B.R., Cornell, D.W.: On coupling multi-systems through data sharing. *Proceedings of the IEEE 75*, 5, 573-587, 1987
YYF85	Yen, W.C., Yen, D.W.L., Fu, K.: Data coherence problem in a multicache system. *IEEE Trans. Computers 34*,1, 56-65, 1985.