Trends in Distributed and Cooperative Database Management

K. Küspert
IBM Heidelberg Scientific Center
Advanced Information Management Dept.
Tiergartenstr. 15
D-6900 Heidelberg, Germany

E. Rahm
University of Kaiserslautern
Dept. of Computer Science
Erwin-Schrödinger-Str.
D-6750 Kaiserslautern, Germany

1. Introduction

In the past, database management systems (DBMS) could roughly be subdivided into two major classes which one might call *PC/workstation DBMS* and *(centralized) mainframe DBMS.*

PC/workstation DBMS provide a set of data management functions in a stand-alone fashion as a single-user system on a personal computer or workstation. There is no real data exchange between a PC/workstation DBMS and other data management systems, remote users, or DBMS at another place. Some PC/workstation DBMS provide limited functions to extract data as a "snapshot" from a remote database to process these data locally on the PC/workstation; there is, however, usually no way to propagate changed data back from the PC/workstation to the remote database at the end. Moreover, PC/workstation DBMS are not able to provide an integrated view of the data in a large organization.

Centralized mainframe DBMS run on a large computer in a multi-user environment and are usually rather elaborated w.r.t. their functionality (concurrency control, database recovery, authorization, etc.). Data management capabilities are centralized on a single computer, and database access is done from "unintelligent" terminals, i.e. from rather simple devices without much internal computing power of their own. On a mainframe DBMS, data integration and data exchange between different users is basically no problem since all the data reside in the same database. On the other hand, the users (sometimes hundreds or thousands) of a mainframe DBMS strongly depend on the availability of the computer and the related system software and on the response time of the system which might be rather unpredictable in many cases. It is therefore quite impossible, for instance, to run a real-time application (CAD application, simulation of a technical process, etc.) in a multi-user environment on a mainframe DBMS since the response time will vary drastically depending on the actual system workload. If the applications (for instance in banking, airline reservation, etc.) get larger and larger and thousands of termi-

nals are connected to a single computer system and DBMS, a severe bottleneck might occur which cannot always be removed just by installing a more powerful computer; in many cases, the specific installation will already be at the upper end of the scale of available computing power. Moreover, if the computer system in such an environment is not available for a while, all users and their applications are directly affected and cannot continue their running operations. Finally, centralized mainframe DBMS are not able to reflect the organizational structure of large companies very well: on the one hand, data integration is of course still a must (to enforce data consistency, etc.) but, on the other hand, the "owners" of the data (divisions, departments, etc.) might be spread all over the world so that there are good reasons to spread their data as well instead of having them all on a single centralized site.

To cope with these shortcomings of traditional PC/workstation and centralized mainframe DBMS, new concepts have been elaborated which go in various directions.

One approach is the idea of *database sharing*. Database sharing intends to solve the problem of limited growth in computer performance (no larger single computer available) as well as the problem of limited availability of a single computer system (in case of a system failure, computer operation and database access are completely down for some time). In the database sharing approach, a hardware and operating system environment is set up where several computing systems (usually mainframe computers) are connected in a way that

- each of these computer systems runs its own DBMS code and has its own main storage and database buffers,
- these computer systems and their DBMS also share the same data set (database files) on disk,
- they are coupled via some high-speed communication line (such as channel-to-channel) to ensure fast data exchange between any two systems.

From the database user's point of view, database sharing provides a single system image since the user needs not be aware of the computer where the database requests are actually processed. With such an overall system architecture, more computers can easily be added if the computing power of the existing ones is not sufficient anymore to meet the application demands. Hardware and software failures of single components (a single computer) do not necessarily affect system availability since the surviving "n-1" computers can continue their operation and take over the workload of the system which has failed. Load balancing (transaction scheduling), synchronization, and recovery are important issues in database sharing and a lot of research has already been done to come up with suitable solutions.

In a *distributed DBMS*, each node and each DBMS has its own data on disk; data are partitioned, i.e. there are - in contrast to database sharing - *no shared data* (files) on disk. The nodes are usually located at remote sites so that there are also no fast communication lines available between these nodes. In a distributed DBMS, similar to a database sharing environment, a single system image may be provided, i.e. the place (node) where the data are stored may be fully

transparent from a user's point of view. Therefore, the data can be stored at a place where they are most frequently needed (at a specific division or department within a large organization) but are still accessible from any other place within that organization as well. Data partitioning can be done in different ways and on different granularities, and data can also be replicated in order to speed up processing. Prominent problems which must be solved are related to query processing and optimization (how to retrieve the data from these different places in an efficient way, how to perform join operations efficiently, etc.) and update processing (how to materialize updates at different places (nodes) in a way that the data are still consistent after a system crash or some other kind of failure).

Workstation-server DBMS are another approach to distributed and cooperative database management. The basic idea of workstation-server DBMS is the use of many workstations (e.g. in engineering applications) with local processing power and a local DBMS which are linked to a server machine (usually a mainframe) with a server DBMS. For the server DBMS, any of the above implementation concepts may be used (conventional centralized DBMS, database sharing approach, distributed DBMS). Workstation *autonomy* and *very fast local data processing* at the workstation are important issues in that environment. The data which shall be processed by a user at a workstation are *checked-out* from the database server and sent to a workstation where local processing is done. Local processing may last for days and weeks, for instance if a large engineering design process must be done. *Long locks* are set on these data to ensure that the data integrity cannot be violated by parallel users at other workstations. At the workstation site, there is a close interaction and very frequent data exchange between the user (his application program) and the local DBMS. The local DBMS must be designed in a way to support that kind of interaction very efficiently. Concepts like an "object cache" are an appropriate solution to speed up data access from an application program. A workstation-server DBMS must also support efficient check-in processing to propagate changed data back from the workstation to the server. Mechanisms which support a *tight cooperation* between database server and workstation turned out to be a very practicable solution to achieve that goal.

The main part of this paper is organized as follows: In Section 2, concepts and implementation of the database sharing approach will be discussed. Section 3 deals with workstation-server database management systems, and in Section 4 distributed database management will be addressed. Finally, Section 5 will give a summary and an outlook together with some concluding remarks.

2. Database Sharing

One approach to overcome the performance and availability limitations of centralized DBMS is database (data) sharing. In a database sharing system, the transaction workload is processed on multiple locally coupled computing systems (usually mainframe computers) that have direct physical access to the entire database on disk (therefore, this approach is also known as "shared disk"). Each of the computer systems runs its own copy of the operating system and DBMS,

and has its own main memory and database buffers. Communication between different nodes typically takes place by means of message passing over a high-speed interconnect such as channel-to-channel adapters. An alternative to such loosely coupled systems is the use of shared semiconductor stores for data exchange (close coupling) to avoid the communication overhead associated with message passing.

A main advantage of database sharing compared to other distributed architectures (e.g. data partitioning, see Section 4) is that there is no need to physically partition and allocate the database among the systems. This results in an increased flexibility for load balancing because every transaction or database operation can be completely executed at any system since each node can directly access the entire database. For instance, it is possible to allocate complex ad hoc queries and short on-line transactions to separate systems to avoid resource contention on CPU and memory between these conflicting workload types. In a data partitioning system, on the other hand, a database operation typically has to be executed where the data reside in order to limit the communication overhead. Thus the workload allocation is mainly determined by the (physical) data allocation leaving little freedom for dynamic load balancing, e.g. to avoid overloading of some processors. Adding a processor in data partitioning systems is also cumbersome since it requires to reallocate the database in order to utilize all systems. The migration from a centralized to a multi-system environment is therefore easier for database sharing since existing databases need not be changed.

2.1 Technical Problems

To take full advantage of the database sharing architecture, a number of technical problems have to be solved, notably in the areas of concurrency and coherence control /Ra88b/, workload allocation, and recovery:

- In loosely coupled database sharing systems, inter-node communication is required for concurrency and coherence control. Concurrency control is needed to synchronize the accesses to the shared database and to enforce global serializability. Coherence control has to deal with the so-called buffer invalidation problem. This problem arises since every system caches pages from the shared database in its database buffer to limit the number of disk I/O's. The resulting replication of pages in main memory also permits multiple systems to read the same data concurrently. On the other hand, modification of a page in one database buffer makes all copies of that page in other buffers (and on disk) obsolete. Coherence control has to make sure that these buffer invalidations are either avoided or detected and that all transactions get access to the current versions of database objects. The number of messages for concurrency and coherence control has to be kept as low as possible to reduce the communication overhead and to limit transaction deactivations due to remote requests.

- Workload allocation is responsible for distributing the transaction workload among the processors. This transaction routing should not be statically determined by a fixed allo-

cation of terminals and/or programs to nodes, but should be automatic and adaptive with respect to changing conditions in the system (e.g. overload situations, node crashes, etc.). Effective workload allocation schemes do not only aim at achieving load balancing to limit resource (CPU) contention, but also at supporting efficient transaction processing with a minimum of inter-system communication or I/O delays. For this purpose, so-called affinity-based routing schemes should be employed that assign transactions with affinity to the same database portions to the same node. (For typical on-line transaction types, the database reference pattern is generally known from previous executions.) This results in improved locality of reference that can be utilized by suitable concurrency control schemes to reduce the number of synchronization messages (see Section 2.2). Furthermore, hit ratios are improved and the frequency of buffer invalidations can be limited.

Crash recovery and media recovery are the major recovery forms that require new solutions for database sharing. Crash recovery for a failed node has to be performed by the surviving nodes in order to provide high availability. In general, lost effects of transactions committed at the failed node have to be redone (REDO recovery) while modifications of in-progress (failed) transactions may have to be undone. Special recovery actions may be necessary to properly continue concurrency and coherence control, e.g. reconstruction of lost control information. Media recovery may require the construction of a global log where the modifications of all nodes are recorded in chronological order.

These problems have been addressed in several existing database (disk) sharing systems (e.g. IMS Data Sharing, DIGITAL's VaxCluster, and Computer Console's Power System) and various research projects. In the next two subsections (2.2 and 2.3), we review the major solutions for concurrency and coherence control in loosely coupled database sharing systems. In Section 2.4, we then discuss the realization of a closely coupled system that utilizes a shared and non-volatile semiconductor store to improve performance. A detailed treatment of workload allocation and recovery is beyond the scope of this paper, but can be found in two recent reports by one of the authors /Ra89a,b/.

2.2 Concurrency Control

Various locking schemes and optimistic concurrency control (OCC) methods have been proposed or implemented for synchronization in database sharing systems. The appeal of OCC is that only one remote request per transaction may be needed for concurrency control, namely for validation at the transaction's end. On the other hand, the amount of wasted work due to transaction restarts (required to resolve concurrency conflicts) can be excessive for transaction workloads of moderate or high conflict probability. This basic trade-off could be confirmed in detailed, trace-driven simulations of various central and distributed OCC protocols with different conflict resolution strategies /Ra88a,b/. It turned out that locking schemes (described below) could obtain comparable performance for read-intensive workloads, but were clearly superior with a higher share of update transactions and in the presence of hot spots. Since locking is also

the method of choice in all commercially available DBMS, we exclude OCC from further consideration in this paper. Rather, we are going to describe a central and a distributed locking approach together with applicable optimizations to reduce the number of remote (global) lock requests. Finally, we briefly discuss the concurrency control methods employed in existing database sharing systems.

2.2.1 Central Lock Manager (CLM)

In the central locking protocol, global locks are processed by a CLM running on a designated node. In the simplest form, every lock request and release is forwarded to the CLM node. This results in two messages per lock request which is not acceptable for high performance transaction processing. Batching of messages reduces the communication overhead, but at the expense of increased delays for the synchronous lock requests and thus increased response times. (Increased response time implies longer lock holding time and thus higher lock contention.) To make the CLM approach more viable, two other techniques can be incorporated that utilize locality of reference and are able to reduce both the communication overhead and response times:

- So-called read optimization /Ra86, Ra88a,b/ allows multiple nodes at the same time to grant and release read locks for a database object locally without contacting the CLM. The first read access to an object O in a node has to be granted by the CLM. If no write lock request is known at the CLM at this point in time, the CLM assigns a so-called read authorization for O to the requesting node. This read authorization gives the node the permission to process all further read lock requests and releases for O locally, thus reducing the number of synchronization messages and response time delays. To make full use of this idea, read authorizations generally are held beyond the transaction's end (in contrast to regular read locks) to allow other transactions to read the respective objects without lock delay. Thus, the effectiveness of this technique increases with increasing locality of read accesses. Write accesses, however, may suffer from this technique since a write lock cannot be granted until the CLM has revoked all read authorizations.

- A similar concept, called sole interest, can be applied to grant an authorization for a local synchronization of read and write requests (write authorization). Such an authorization is assigned to a node when it requests a lock at the CLM and no other node has issued a lock request for the same page ('sole interest'). Of course, a write authorization can be assigned only to one node at a time, and has to be revoked by the CLM as soon as any other node requests a read or write lock for the same object. If a read request causes the sole interest revocation, the write authorization is degraded into a read authorization. Otherwise the write authorization of the current owner is given up, and assigned to the requesting node (if there are no waiting requests from other nodes).
  The sole interest concept pays off only if more lock requests can be locally satisfied than sole interest revocations occur. This is because four messages are required for a lock request causing a sole interest revocation, compared to two messages without sole interest concept. In contrast to the read optimization, the effectiveness of sole interest depends on the amount

of node-specific locality of reference requiring that different nodes should reference different portions of the database. Affinity-based transaction routing is necessary to support this requirement.

Both techniques can be applied at different levels of the object hierarchy, e.g. database files and pages or record types and tuples. For instance if a node holds a write (read) authorization for an entire file, all (read) lock requests against pages of this file can be locally synchronized. The complexity of such a hierarchical scheme, however, is substantially higher compared to the case where read optimization and sole interest are restricted to the smallest concurrency control granules. Also, for "important" files or record types to which a substantial share of the database references is directed, it is generally unlikely that only one node has interest or that only read references are issued for longer periods of time. Rather, thrashing-like situations with only short-lived assignments and frequent revocations of read/write authorizations may occur that cause more additional messages than are saved.

2.2.2 Primary Copy Locking (PCL) Approach

In this distributed scheme, the database is divided into logical partitions and each node is assigned the synchronization responsibility (or primary copy authority, PCA) for one partition. Lock requests against the local partition can be handled without communication overhead and delay, while other requests have to be directed to the authorized processor holding the PCA for the respective partition. In order to reduce the number of remote lock requests, the PCA and workload allocations should be coordinated such that transaction types are generally allocated to the node where most data references can be locally synchronized. In addition, a read optimization can also be employed for the primary copy scheme where the read authorizations are assigned and revoked by the PCA Lock Manager. This permits a local read synchronization of objects belonging to the partition of another node. Details of the protocol can be found in /Ra86/.

At first sight, the need to determine a PCA allocation could be considered as a disadvantage of PCL compared to the CLM approach. This is not the case, however, since both schemes have to coordinate workload allocation and concurrency control to limit the number of remote requests which is easier achieved for PCL. In the CLM scheme, sole interest assignments are dynamically assigned and revoked if more than one node wants to access a given data object. Thus these assignments can be highly unstable (in particular for hot spot objects) making it difficult and expensive to determine for an incoming transaction where it can be processed with few remote lock requests. The PCA allocations, on the other hand, are stable and do not change when multiple systems need access to the same data. This makes it easier to achieve an affinity-based transaction routing such that the number of remote lock requests can be kept low. In general, it is even possible to use an efficient table-driven approach where the assignment of transaction types to processors is determined by a routing table which needs only be adapted after significant changes in the load profile or system state (e.g. processor crash). PCL also has no analo-

gous disadvantage to the expensive revocations of write authorizations in the CLM scheme. Simulation studies have confirmed the problems of the sole interest concept that resulted in inferior overall performance compared to database sharing systems employing the PCL approach /Ra88a,b/.

Given that the PCA allocation is a special form of data allocation, the question arises what advantages remain for database sharing with PCL compared to data partitioning. The first point is that the PCA allocation is only a logical data assignment (represented by internal control structures) which can therefore be more easily adapted to changing workload/system conditions than the physical data allocation for data partitioning. Secondly, there is still a high potential for load balancing since the PCA allocation only determines the distribution of lock overhead while the largest part of a transaction can be processed on any node. In data 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. (This holds at least for the simple operations prevalent in on-line transactions.) Finally, the performance of database sharing with PCL depends to a lesser degree on how well the database can be partitioned to reduce the number of remote requests. This is because main memory caching and read optimization efficiently support concurrent read accesses to the same data in multiple systems. In data partitioning systems, typically all read and write accesses to an object take place at the "owner" node.

2.2.3 Concurrency Control in Existing Database Sharing Systems

So far the PCL approach and the read optimization (which is essentially applicable to all locking protocols for database sharing) have only been implemented in simulation systems, but not in prototypes or commercially available database sharing systems. A CLM scheme is used in the database sharing system of Computer Console /WIH83/ and in the Amoeba prototype /Tr83, Sh85/. They rely on a sole interest concept for coarse granules (files, record types) for reducing the communication overhead, although the effectiveness of this approach is questionable (see above), in particular if no appropriate strategies for affinity-based transaction routing are provided. IMS Data Sharing /SUW82/ uses a token ring protocol for lock processing (called "pass-the-buck") where remote lock requests can be batched together with the token to reduce the communication overhead. The protocol has been restricted to two systems since the turnaround time per global lock request (token circulation time) grows proportionally with the number of nodes.

A distributed lock protocol is employed in DIGITAL's VaxCluster /KLS86/. It has some similarities with the PCL approach since for every data object there is a "master" node being responsible for synchronization. The mastership, however, is not predetermined like the PCA allocation, but is dynamically assigned to the node that issues the first lock request for an object. The current mastership distribution is stored in a directory that is partitioned among all systems according to a hash function. This indirection results in up to four messages per lock request

(two to determine the master node from the directory, and two for the lock request itself), compared to at most two messages for PCL.

In addition, mastership assignments are unstable (similar to sole interest assignments) and therefore difficult to consider for transaction routing (affinity-based transaction routing is not supported in VaxCluster configurations).

The TPF (Transaction Processing Facility) operating system kernel supports disk sharing for up to eight systems /Sc87, TPF88/. A rudimentary form of locking is performed by the shared disk controllers that maintain a lock table in their memory. 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. 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.

2.3 Coherence Control

Coherence control has to ensure that every transaction sees only up-to-date data despite the fact that cached data may be invalidated by update transactions running on other systems. The solution of this problem depends on the concurrency control granularity as well as on the strategy for update propagation to disk:

- Since the data replication in the buffers takes place on page level, concurrency control on smaller granules becomes more complex than in centralized DBMS. Record-level concurrency control would permit that different records of the same page are concurrently modified in different buffers (systems). As a result, none of the buffers would hold a completely up-to-date page and writing out these pages could lead to lost updates. Since merging the modifications is expensive (or even impossible), at least writes between different systems generally have to be synchronized on page level. Record-level concurrency control may be used for read accesses and concurrent read/write accesses within the same system.

- A FORCE scheme /HR83/ for update propagation requires that all modified pages are forced to disk before the modifying transaction commits. This approach is often unacceptable for performance reasons since it causes a high I/O overhead and significant response time increase for update transactions. Nevertheless, in contrast to centralized DBMS all existing database sharing systems still employ the FORCE scheme thus sacrificing performance for the sake of a simplified crash recovery /Ra89b/ and coherence control (the most recent page version can always be found on disk with FORCE). We feel that NOFORCE should also be supported for database sharing since the extra problems can be solved with reasonable effort.

For NOFORCE the permanent database on disk is generally obsolete so that one has to keep track of where the most recent version of a modified page can be obtained. Instead of reading the page from disk, a page request may have to be sent to the system holding the current page version in its buffer. The page can then be returned to the requesting system either directly over the communication lines or across the shared disk. With a high-speed interconnect, the direct page transmission is faster by at least a factor of 10.

In the following, we outline the three major approaches to coherence control in database sharing systems:

- broadcast invalidation,
- on-request invalidation,
- and avoidance of buffer invalidations by retention locks.

The first approach is applicable to any concurrency control method, but introduces the greatest overhead. On-request invalidation is compatible with the CLM and PCL approaches, while retention locks are limited to schemes that apply a sole interest concept, such as the CLM scheme. The discussion assumes that locking takes place on page level.

2.3.1 Broadcast Invalidation

This simple approach, in combination with FORCE, is used in most existing database sharing systems. To detect buffer invalidations, a broadcast message is sent at the end of every update transaction indicating which pages have been modified. Invalidated pages can thus immediately be removed from the database buffers. On the other hand, the write locks of the update transaction must not be released until all systems have acknowledged that they have processed the broadcast message and discarded the invalidated page copies (otherwise, access to obsolete data would be possible). Thus, response time is increased as well as a substantial communication overhead is introduced that grows with the number of systems.

NOFORCE requires additional provisions in order to provide a transaction with the most recent page copies. For this purpose, a special table can be maintained in every system indicating for all (recently) modified pages where the latest modification has been performed and thus from where the current page version can be requested. These tables are maintained without extra communication overhead by using information from the broadcast messages. By periodically broadcasting which modified pages have been written to disk, the number of table entries can be limited. These notifications can be piggy-backed to the broadcast invalidation messages.

2.3.2 On-Request Invalidation

This approach uses extended information in the global lock table which allows the lock manager (CLM or PCA Lock Manager) to decide upon the validity of a buffer page together with the lock request processing. Since a lock has to be acquired before a (cached) page can be accessed, obsolete page copies can be detected without any additional communication, a main advantage

compared to broadcast invalidation schemes. The information needed to detect buffer invalidations (e.g. page sequence numbers) is updated for every modification, together with the release of the write lock.

For NOFORCE, it can additionally be recorded in the global lock table from which system the current version of a modified page can be requested. A different approach is possible for PCL by always providing the PCA nodes with the most recent version of the pages from its partition /Ra86/. No extra communication is necessary for this by sending modified pages to the PCA node together with the message required for releasing the write lock. Thus the most recent page version can always be obtained from the PCA node (or from disk in the case the page does no longer reside in the PCA node's database buffer). This permits a combination of lock requests and page requests because the PCA node can use the message to grant a lock to an external transaction for transmitting the respective page as well. In this way, extra messages are avoided not only for detecting buffer invalidations, but also for exchanging modified pages between different systems. An added advantage is that buffer invalidations are now only possible for cached pages belonging to the partition of another node. Ideally, most modifications are performed at the PCA node thus limiting the number of buffer invalidations and page transfers.

An on-request invalidation scheme based on page sequence numbers is used in DIGITAL's VaxCluster /KLS86/ in combination with FORCE. In /Ra86/, an alternative is described that uses so-called invalidation vectors to detect buffer invalidations so that there is no need to store version numbers in every page.

2.3.3 Avoidance of Buffer Invalidations

Buffer invalidations are only possible for cached pages that are modified at another system. 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 anymore.

A better approach is to retain the pages in main memory but to protect them from invalidation by special retention locks. This approach is particularly attractive for the CLM scheme since it can be combined with the realization of a sole interest concept and read optimization on page level. In this case, we have two types of retention locks represented by a sole interest (SI) assignment or a read authorization (RA). The use of these locks can be characterized as follows:

- For every cached page either a retention lock or a regular transaction lock must be held at the respective node.

- For modified pages not currently locked by an active transaction, a SI 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 pages are protected by an RA retention lock which can be held by multiple systems concurrently. In addition, RA guarantees that no system holds a write lock or SI retention lock thus permitting a local synchronization of read accesses (read authorization).

- If a lock request has to be be processed by the CLM, it may be necessary to revoke incompatible retention locks before the lock can be granted. Before a retention lock is released, the corresponding page is purged from the buffer to avoid its invalidation. A modified page (SI revocation) is either written to disk or directly transferred to the requesting system if it is going to obtain a sole interest assignment. In the latter case, a separate page request is avoided since the page exchange is combined with the SI revocation.

- 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. In this case, the associated retention lock should voluntarily be released to limit the number of revocations and lock table entries.

Although we cannot go into further details, it should have become clear that this approach avoids buffer invalidations without introducing extra messages in addition to the ones needed for revocation of read and write authorization.

2.4 Use of Shared Semiconductor Stores

A prime objective in the design of a loosely coupled database sharing system is to reduce the number of remote requests for transaction processing. This is because the communication overhead associated with message passing reduces the effective CPU utilization and thus the achievable transaction rates. In addition, response times and thus data contention are increased. Data contention may become a performance bottleneck if lock conflicts prevent full utilization of all processors. This danger grows with the CPU speed and the number of CPU's as higher multiprogramming levels have to be applied to fully utilize the added capacity.

Closely coupled systems aim at a more efficient cooperation between systems, e.g. by utilizing shared semiconductor stores. In contrast to tightly coupled multiprocessors with shared main memory, however, the computers connected to the shared store are autonomous (i.e. they have their own main memory and copy of operating system and DBMS) to improve failure isolation. Still, access to such a store should be very fast (e.g. a few microseconds) to permit a synchronous access, i.e. without releasing the CPU to avoid a process switch (overhead). This is in contrast to disk caches or solid-state disks that offer a disk-oriented interface (channel commands) with access times of about 2 ms per page.

A so-called extended storage appears to be more appropriate. In current mainframes (e.g. IBM 3090), it is used as a fast paging device which is controlled by the operating system. Access

times in the order of 50-100 microseconds per page permit a synchronous access. Although the extended storage is currently volatile and cannot be shared by multiple systems, we expect that these limitations can be resolved in the near future. Providing non-volatility is comparatively easy, e.g. with a battery backup or uninterruptible power supply. More critical is the design of the access interface and the controllers of a shared store. Providing a low-level interface, as in the current extended storage, simplifies the hardware and facilitates fast access times. In this case, use and administration of the shared data is mainly up to the software (operating system, DBMS) in the accessing systems. On the other hand, putting more functionality (e.g. global lock management) into the storage controllers makes the hardware more complex and error-prone and limits the usefulness of the shared store to special applications. In addition, access times are higher and waiting times at the controller may no longer permit a synchronous access.

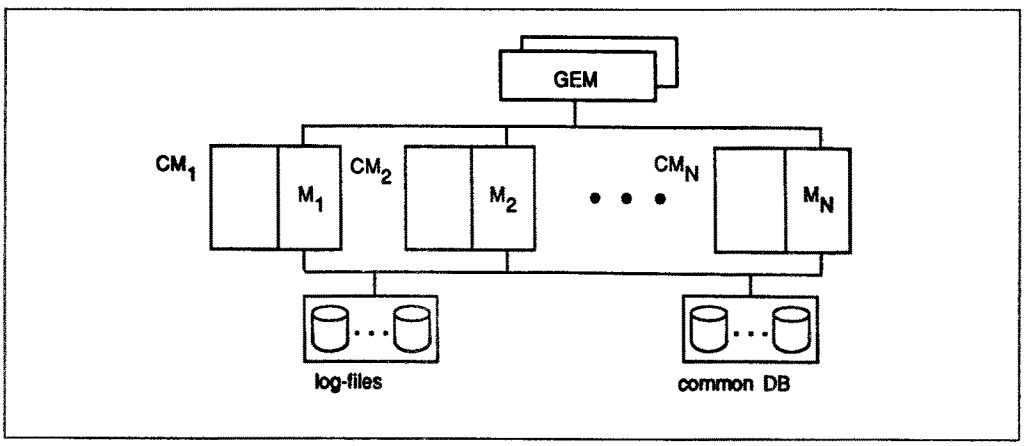

Figure 2.1: Database Sharing System with Global Extended Memory

In /BHR90/, a special store called global extended memory (GEM) has been investigated for use in database sharing systems. This shared store (*Figure 2.1*) is assumed to be non-volatile and offers a simple interface with fast, synchronous access. To enhance fault isolation, data in the GEM cannot directly be manipulated by the accessing systems but has to be read into main memory and written back after modification. The access granules are either entire pages or smaller units (entries) which may be used to realize simple data structures. Besides of reading and writing these granules, hardware instructions like compare&swap are supported to synchronize concurrent GEM accesses. To deal with GEM failures, duplicate data storage in independent GEM storage units is possible (analogous to disk mirroring).

Despite the simple access interface, GEM can be utilized in database sharing systems for various tasks. One possibility is to store a global lock table in GEM to permit every node to decide upon whether or not a lock request can be granted. With appropriate design of the lock protocol (e.g. only a reduced lock/coherence information on a per-system basis needs to be maintained in

GEM), locks may be granted in a few microseconds so that negligible overhead is introduced for global concurrency control. The non-volatility of GEM significantly speeds up write I/O's for database and log pages. By maintaining a global database buffer as well as local and global log files in GEM, a dramatic reduction in I/O delays and I/O overhead can be expected. The most general application of GEM, not limited to database sharing, is to use it for inter-system communication such that all messages are exchanged across the GEM. For this purpose, a "message" first has to be written to GEM and the destination system is notified by an interrupt indicating the GEM location of the message. The message is then read by the destination system from the specified address. This message exchange may incur substantially less overhead than traditional message passing over communication lines, provided the interrupt handling can be kept inexpensive.

A preliminary performance evaluation of GEM usage is reported in /BHR90/. A simulation study has been conducted that compares the performance of loosely coupled database sharing systems with closely coupled configurations using GEM. In both cases, the PCL protocol has been employed for concurrency/coherence control, but with a message exchange via GEM in the closely coupled configurations. In addition, it was assumed that the entire database and all log files are GEM-resident. The GEM configurations achieved significantly better response times since I/O and communication delays were largely eliminated. In addition, the CPU's could be utilized at very low multiprogramming levels since very few transaction delays had to be overlapped. As a consequence, lock contention was almost negligible in contrast to the loosely coupled configurations where much higher concurrency levels had to be applied. The reduced lock contention decreases the need for fine-granularity locking and facilitates vertical growth (faster CPU's) as well as horizontal growth (scalability). Horizontal growth is also supported by the reduced communication overhead compared to the loosely coupled configurations.

2.5 Summary

Database sharing is a locally distributed architecture that offers a high potential for achieving high transaction rates, high availability, and horizontal growth. Although existing database sharing systems still fall short to fully utilize this potential, the techniques for better solutions are available. Key factors in the design of high performance database sharing systems include the use of a NOFORCE strategy for update propagation to disk, affinity-based transaction routing, integrated solutions to concurrency and coherence control, and the use of a fast communication system. For loosely coupled configurations, we recommend the primary copy locking scheme together with on-request invalidation for coherence control. The central Lock Manager approach (using retention locks to avoid buffer invalidations) may be appropriate in closely coupled database sharing systems where the global lock table is maintained in a shared semiconductor store. Shared, non-volatile semiconductor stores like a GEM promise a significant reduction in I/O and communication delays and can thus facilitate vertical and horizontal growth. We expect such stores to be used for high-volume transaction processing in centralized and closely coupled systems during this decade.

3. Workstation-Server Database Management

In the following, we will explain why workstation-server database management is a logical consequence of the shortcomings of fully *centralized database management*[1] on the one hand (e.g. "pure" mainframe DBMS) and fully *decentralized database management* (PC/workstation DBMS) on the other hand (Section 3.1). We will then address the requirements of "non-standard" database management applications w.r.t. the integration of workstations into the overall system scenario (Section 3.2). The essential properties of a workstation-server DBMS will be explained, and the characteristics and implementation of a specific prototype system - the Advanced Information Management Prototype (AIM-P) /Da86, Pi87, DL89/ - will be discussed (Section 3.3). Finally, Section 3.4 will give a short summary.

3.1 Why Workstation-Server Database Management?

Traditionally, database management has been done in a (logically) centralized fashion: The DBMS was located on a large mainframe computer[2], and data access was done by the users from numerous "unintelligent" terminals without much processing power of their own. This scenario can also be called a server- or host-based solution for database management (*Figure 3.1*). The user in that scenario is fully dependent on the availability and response time of the DBMS and the underlying hardware and system software.

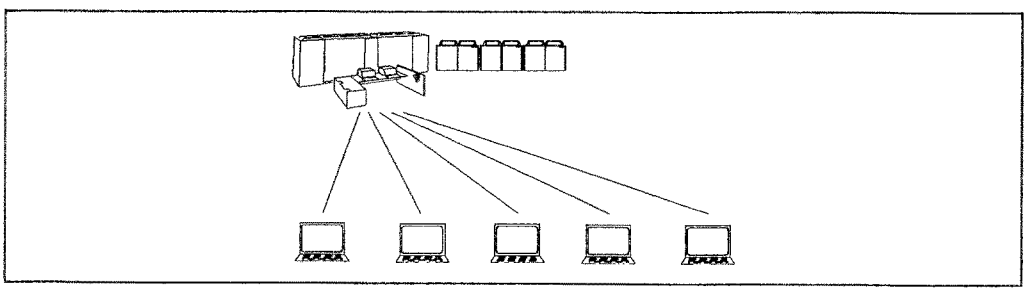

Figure 3.1: Server-Based Solution for Database Management

To gain more independence from a centralized system with its overloading and casual failures affecting *all* users, a workstation-based solution was often seen as an alternative (*Figure 3.2*): The user has all his data locally on the workstation. He does not depend anymore on the availability and response time of a remote system with hundreds or thousands of users. For a high

---

[1] The term "centralized database management" stands for a *logically* centralized system, i.e. a single system image. We do not necessarily mean physical centralization (*one* computer).

[2] ... or on *several* mainframe computers in case of database sharing or distributed DBMS.

performance computer graphics application or a computer simulation, for instance in engineering, such a local DBMS may provide a proper basis for data management with reasonable performance. A disadvantage of a fully decentralized environment, even if the workstations are linked via a network, is that data integrity cannot be easily enforced, that inconsistent and incompatible database schemes and data instances may occur, and that integrated data evaluation cannot be accomplished anymore.

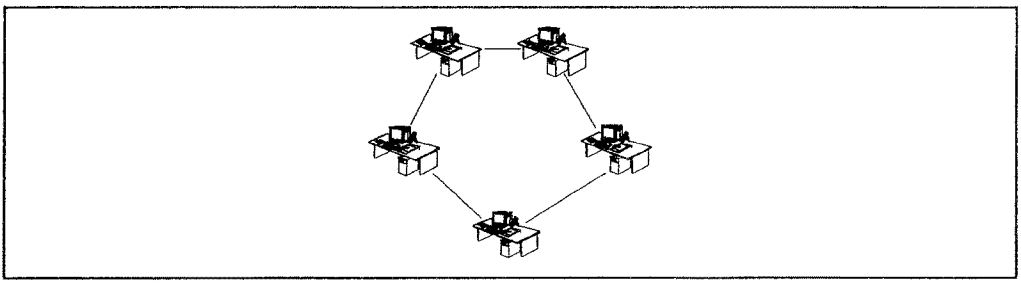

Figure 3.2: Workstation-Based Solution (Network of Workstations)

As a consequence, techniques have been investigated to find a solution so that data integrity can still be enforced by a centralized system with powerful data management capabilities based on a mainframe computer *and* actual data processing can be done locally and rather autonomously on a workstation in order to meet ambitious performance and availability requirements. Such a *workstation-server-based solution* may encompass different levels of centralization and integration: A group of workstations may be integrated, for instance, via a departmental computer. All departmental computers, in turn, may be further integrated via a large mainframe computer. This scenario is shown in *Figure 3.3*.

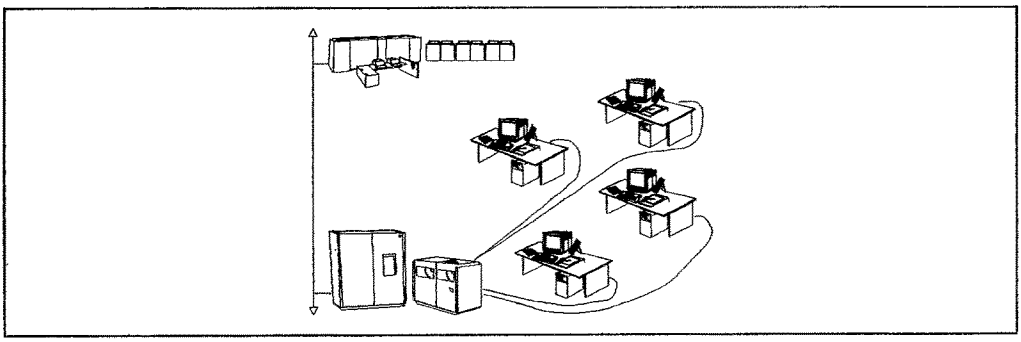

Figure 3.3: Workstation-Server-Based Solution

Workstations may of course also be directly linked to a server without a departmental computer in between. Long term data management is done on the server (server database)[3] whereas short term data management is done on the workstations (workstation database).

Data which shall be processed on a workstation are extracted from the server database and transferred to the workstation where they are put into the workstation database. The workstation database now contains a local *copy* of selected data items from the server database. The local data may be kept on the workstation for days or weeks, depending on the application characteristics. The data are changed locally and the changes are kept as "private data" on the workstation as long as the data are still inconsistent and should not be shown to other users (uncommitted data). Finally, when a new logically consistent state of the data has been reached, the changed data are transferred back into the server database where the changes become visible for other users as well. This is a typical scenario for database usage in a workstation-server environment.

A workstation-server DBMS cannot be implemented just by taking a client-server DBMS or coupling some existing DBMS to run on the server and on the workstation, respectively. Rather, a *tight cooperation* between the server DBMS and the workstation DBMS is a must to achieve acceptable performance and to provide the expected functionality. These *requirements*, which come from the special needs of "non-standard" database applications w.r.t. workstation-server integration, will now be discussed in more detail.

3.2 Workstation-Server DBMS: Requirements and Solutions

On a more technical level, the following *list of requirements* can now be defined for a workstation-server DBMS. To make things more clear, we also sketch the *solutions* to some extent.

3.2.1 Efficient Check-Out and Check-In Processing

Data which shall be processed on a workstation are extracted from the server database and transferred to the workstation. The data thereby become "private data" of the user at the workstation (*check-out* processing). These data must therefore be *locked* in the server database appropriately (see also Section 3.2.2 below). The check-out specification (which data shall be extracted and transferred) can be done via normal database query statements, for instance in SQL. These query statements are embedded in an application program on the workstation and are sent to the server via services of the Application Program Interface (API). Since the workstation user is usually not a database expert, ad hoc queries from the screen (i.e. via an On-line

---

[3] The server database may be a physically centralized, a distributed, or a shared one. The term "mainframe" or "server" therefore denotes a *logically* centralized system which may consist of several computers.

Interface) are of minor interest in a workstation-server scenario. The requested data (may be a large set of tuples) are sent to the workstation where they are stored in a local database table. Since a single check-out may often affect a large amount of data (e.g. if a large engineering object shall be extracted), data transfer to the workstation should be *set-oriented* on a suitable data granule (set of tuples, set of database pages, ...) to minimize the number of interactions between server and workstation.

When the requested data are on the workstation, further processing can be done rather *autonomously*, i.e. the user may continue his work locally even if the server or the communication line are not available for a certain period of time because of a failure or system shutdown. *Autonomy* is one of the essential properties and benefits of workstation-server database management.

When a logical unit of work, like a non-trivial modification of an engineering object, has been finished and the data are logically consistent again, *check-in* can be done to transfer the private data back into the server database. These private data will then become public data again.

Data check-out and check-in can be done on *different levels* within the system hierarchy of a DBMS. We will here only explain the "extreme cases" of these levels of check-out and check-in processing; more details on that can be found in /De86/.

Let us assume that *complex objects* are checked-out and checked-in. These complex objects may consist of data which are stored in different database tables with some interrelationship between these tables, like referential integrity constraints. A scheme for complex object data exchange on *SQL command level* is shown in *Figure 3.4*.

The check-out request is done via SQL commands. Database objects (tuples from the query result) are sent from the server to the workstation where local modifications are done on these data (insertions, updates, and deletions). Finally, when all modifications have been performed, these local modifications are materialized in the server database via another series of *SQL commands* which are sent from the workstation back to the server. A consequence is that the same amount of work (SQL statements) which has been performed on the workstation (SQL command execution) must be done once again during check-in processing. This is in fact a duplication of work which makes check-in processing very expensive.

For that reason, to save time during check-in, data exchange on *page level* can be seen as another (extreme) alternative (*Figure 3.5*).

During check-out, a *set of database pages* covering all the selected data is extracted (on SQL request) and transferred to the workstation - instead of individual tuples in result table format, as we discussed it before (Figure 3.4). The extracted database pages are modified on the workstation (direct update of the page contents) and are transferred back into the server database

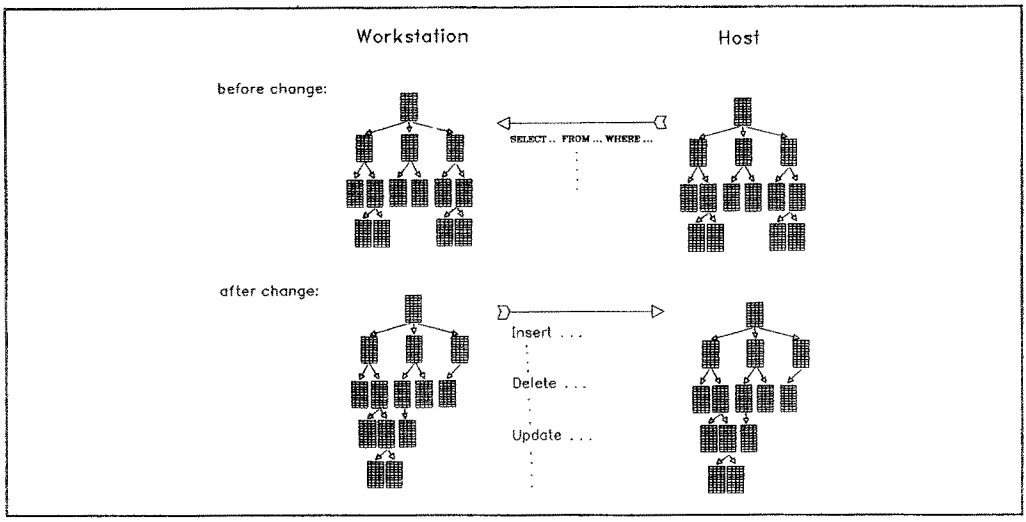

Figure 3.4: Data Exchange on SQL Command Level

at check-in time. Check-out on page level may be more efficient than on tuple level, depending, however, on the actual data distribution (number of selected tuples per database page, etc.). Check-in on page level will usually be more efficient than on SQL level since data are now checked-in on a "very physical level" (pages with all the materialized changes) rather than on a logical level (SQL commands to be executed on the server database). Moreover, a single (modified) database page may contain a large number of changed tuples which had to be modified step by step in case of SQL command level check-in processing. The "delta symbol" in Figure 3.5 (Δ) shall indicate that only the "deltas" (i.e. the changed pages) are sent back to the server for check-in processing whereas the unchanged pages can be discarded on the workstation.

There are many more alternatives (and related problems) for check-out and check-in processing which cannot be discussed here in full detail. Especially the implementation of check-in processing is not an easy task if it shall be done on a physical level (tuple, page) rather than on an SQL command level. Since check-in on a physical level more or less "circumvents" normal SQL services, the implementer will have to cope with problems of (logical) integrity enforcement, index update, catalog (schema) changes, and proper locking granularity. If pages are checked-in via DBMS Buffer or Segment Manager services, logical integrity can of course not be enforced by mechanisms and system components which are located on a higher level in the DBMS hierarchy (Access Path Manager, Record Manager, etc.). The implementer of the check-in mechanism must therefore find a solution to take care of data integrity. One (partial) solution is to *separate* check-out and check-in of "normal" data (primary data) and secondary data like catalogs, access path data, etc. Primary data are then checked-in on page level whereas (changed)

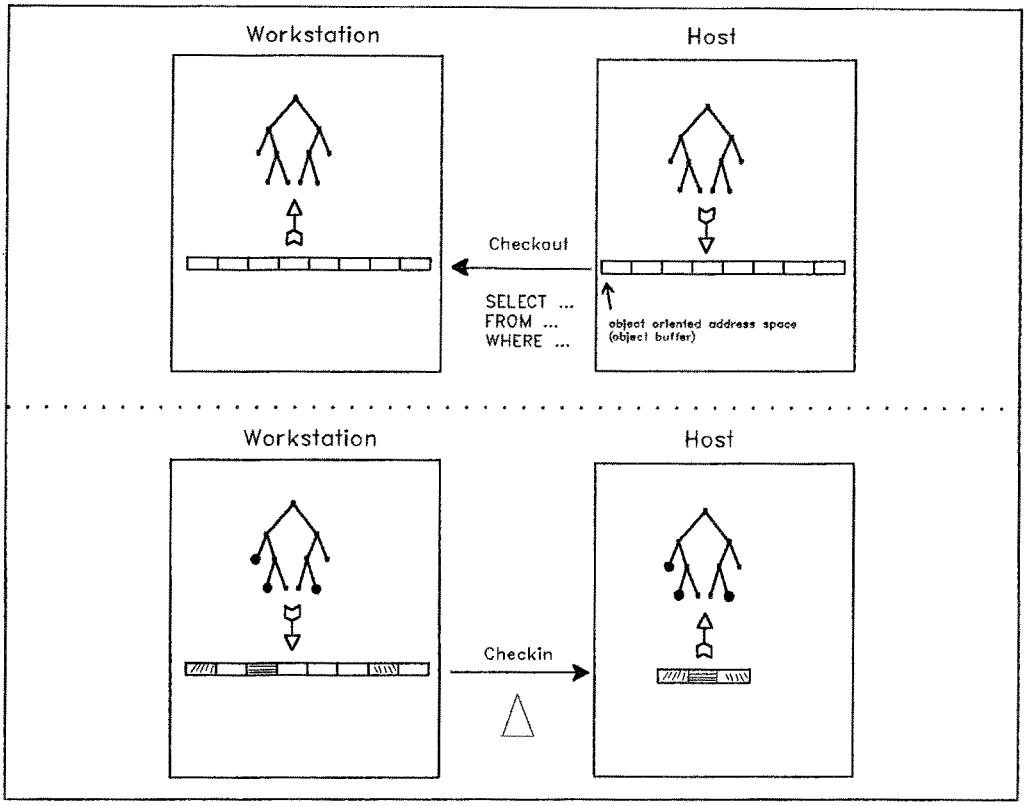

Figure 3.5: Data Exchange on Page Level

catalog and access path data are checked-in on a higher level as catalog or access path update commands. This, however, makes check-out and check-in processing still more complicated since more kinds of data exist and must be distinguished /DO87/.

3.2.2 Long Transactions and Long Locks

Long locks are needed to protect private data on a workstation for an arbitrary (application-dependent) period of time, i.e. for the duration of a *long transaction*. Data are considered as private data after check-out processing when they have been transferred from the server to the

---

[4] Data which are checked-out and transferred to a workstation are of course not physically removed from the server database, i.e. they can, in principle, still be accessed by other (parallel) users, transactions, and workstations. Therefore, some protection (such as appropriate locking) is required to avoid problems of lost updates and other inconsistencies in the server database.

workstation. It should be up to the user - and not up to the DBMS - when these private data become public again. A complex engineering design process on a workstation may keep the affected data privately for days or weeks. During the time between check-out and check-in processing, the related "master copy" of the data on the server must be protected from being changed or deleted by other users.[4]

Traditional transaction management and locking in DBMS are not sufficient to support these long transactions. Traditionally, locks are always "short" and transactions are aborted in case of a system failure or normal shutdown. Short locks are kept in a lock table in (virtual) memory and the lock table contents is lost when the system restarts after a failure or normal shutdown. In fact, there is even no need to write that lock table to non-volatile storage for normal short transactions. Long transactions, on the other hand, must *not* be aborted in case of a failure or normal shutdown: their locks must rather be held over any number of system failures or shutdowns. Therefore, to protect the private data appropriately and to shield the users from each other, long transactions require *long locks* without any fixed upper bound regarding duration. The implementation of these long locks and short locks must be rather different from each other: Because of the requirement of durability, long locks must be written to non-volatile storage (on the server side), for instance into a *database lock table* on disk.

Transaction management must now distinguish between long transactions and short transactions, and different lock tables and lock modes must also be handled. Both long locks and short locks must be observed by parallel users and their transactions. The kind of action to be performed when a lock is encountered must be different, however, for short and long locks and their transactions: If a transaction t_1 encounters a short lock held by another transaction t_2, t_1 will usually be blocked and put into a wait state. It will be resumed when the lock has been released. In case of a long lock, however, blocking t_1 and putting it into a wait state does not make much sense since waiting for hours, days, or weeks is usually not acceptable. Therefore, if a requested object is currently not accessible because of a long lock, the requesting transaction must be *informed* rather than being blocked. It is then up to the requesting user or application program how to react (try again later, access other data, check who is the owner of the locked data, etc.).

3.2.3 Separation of Recovery Unit and Isolation Unit

Traditionally, a database transaction serves both as recovery unit and isolation unit: *Isolation unit* means that parallel users and transactions are fully isolated from each other; database modifications made by a running transaction are not visible to other (parallel) users before end of transaction (EOT). This is usually achieved via locks. *Recovery unit* means that the transaction is the unit of UNDO or REDO processing in case of a failure /Re81/.

In a workstation-server environment a *long* transaction will act as an isolation unit but cannot act as a recovery unit, since the work of days or weeks within a long transaction cannot simply

be aborted in case of a failure. Therefore, in addition to long transactions, short transactions ("normal" database transactions) are still needed as a basis for recovery processing on the server (during check-out and check-in processing) and on the workstation (for local UNDO or REDO processing after a failure).

3.2.4 Fast Processing of Database Objects on the Workstation

The fact that there are usually no parallel users or other "resource consumers" on a workstation is an important reason for establishing a workstation-server configuration. Obviously, data exchange with a local workstation DBMS in single-user mode can be much faster than with a remote DBMS in a multi-user environment. Any data exchange with any DBMS, however, is more or less time consuming since the enhanced functionality of a DBMS is always expensive in usage of system resources. Since many kinds of workstation applications, like computer-based simulation, CAD, etc., are operating under very tight response time restrictions, suitable mechanisms are required to exploit all the functionality of a powerful workstation DBMS *and* to alleviate the performance implications.

At some level of abstraction, a workstation database can be seen as a *large local buffer* (on non-volatile storage) for data which should not be retrieved from and written back to the server database every time. If the workstation DBMS is now augmented by an *object cache* in virtual memory, another stage of buffering has been introduced and can be used for further performance optimization.

The object cache resides between the application program and the workstation DBMS. Data are loaded on application request (object fetch) from the workstation database into the object cache. To save processing time and DBMS calls, data (object) loading into the cache should be done on a set-oriented basis. The data are thereby automatically transformed from the database format (tuple format) into an application-oriented (programming language) format. The data in the object cache can then be directly processed with "normal" programming language statements without further DBMS interaction. Processing of data in the programming language is of course much faster than any DBMS call, be it a local (workstation DBMS) or a remote one (server DBMS). This is especially important for database applications with a large number and high frequency of data fetch operations: Once the data resides in the object cache, numerous DBMS calls with expensive parameter passing and checking, long instruction paths, etc. can be avoided and replaced by simple in-core data addressing.

Object cache management, however, is only simple as long as *retrieval operations* are considered. If the object cache shall be used for data modifications as well and if data modifications shall be done in the cache via normal programming language statements, the problem of *update materialization* must be solved: If updates are done in the object cache without any DBMS interaction or notification, it is quite hard - or might even be impossible - for the DBMS to find out at the end which data have actually been changed and must therefore be written to the da-

tabase. One approach to support that kind of "change detection" could be to force the application program to *flag* all changes appropriately so that the DBMS must only scan the data and look for the flags in order to find out what to do. This, however, burdens the application program with the additional task of "flag management" and substantially complicates the programming task. It is therefore still under discussion how object caching can also be efficiently used in a scenario where data are read *and* changed via an object cache /Ke89/.

3.2.5 Some Open Issues

The above list of requirements (and possible solutions) for a workstation-server database management system is of course still incomplete and there are also still some open issues w.r.t. the functionality of such a system.

One of these issues deals with *transaction management on the workstation*: What do workstation users actually expect from a workstation DBMS w.r.t. recovery management? If the different kinds of log data needed for local transaction management are only kept on the workstation, these data may easily be affected by some kind of major "disaster" (complete loss of data) since an office is usually not a computing center w.r.t data protection and security. In some cases, this could mean that the work of days or weeks gets lost because of some kind of failure. Moving certain log related data (archive copies of the workstation database and/or log files) back to the server may provide a better basis to recover from these kinds of failures. This could be done, for instance, once per day to provide a "safepoint" on the server for recovery processing after a failure.

Another open issue deals with the required *query processing capabilities* on the workstation. The question is whether the full power of SQL should also be available for local processing on the workstation (and not only for check-out processing on the server). As an alternative, an Application Program Interface (API) with restricted processing capabilities on the workstation (just navigation on the data, etc.) might be sufficient for a large class of applications. Answers to these - and other - questions, however, can only be given if there is more experience with workstation-server database management systems in practice.

3.3 Workstation-Server Database Management in AIM-P

The Advanced Information Management Prototype (AIM-P) is a prototype DBMS which has been developed at the IBM Heidelberg Scientific Center /Da86, Li88, DL89/. From the very beginning, AIM-P has been designed with workstation-server processing in mind. AIM-P supports an extended NF^2 (Non First Normal Form) data model with an upward-compatible SQL dialect, called HDBL (Heidelberg DataBase Language), that is able to handle complex and "flat" database objects in a uniform way. AIM-P is an extensible database management system: The user may define his own (complex) data types and his own functions based on these types. User defined data types and functions are an integral part of the AIM-P data model and lan-

guage interface (HDBL). The AIM-P data model, its language, and its extensibility mechanisms have already been described elsewhere /PT85, PA86, Li88/; these discussions shall not be repeated here. Rather, we will concentrate on the overall AIM-P system architecture and on workstation-server related aspects.

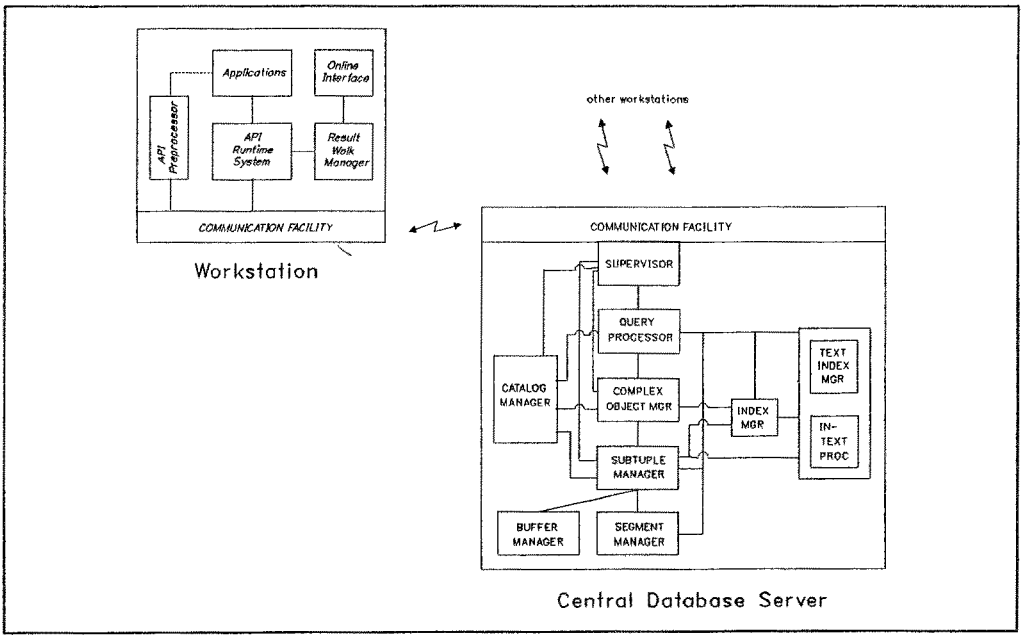

Figure 3.6: AIM-P System Architecture

Figure 3.6 shows the AIM-P system architecture in a workstation-server environment. AIM-P does not offer full SQL (or HDBL) functionality for local database processing on the workstation. There is also no Query Processor for HDBL processing on the workstation (upper left part of Figure 3.6). The workstation DBMS owns a subset of functions of the server DBMS with some additional services which do not exist in the server DBMS. The *API Runtime System*, for instance, which is based on the *Result Walk Manager*, is a specific component of the workstation DBMS without a related server component. The API Runtime System, together with the *API Preprocessor*, implements a cursor interface which is to be used by the *Applications* on top. In contrast to the principles of SQL, the cursor concept of AIM-P is a hierarchical one, i.e. cursor hierarchies are used to process complex objects of the extended NF^2 data model /ES88/.

Retrieval and update processing of database objects on the workstation is done via the following *steps* (for a more detailed discussion see /KDG87, KG89/):

1. Complex objects which shall be processed on a workstation are selected from the server database via an HDBL query statement. The query statement, which is embedded in an application program on the workstation, is sent to the server DBMS and the requested data are extracted from the server database.

2. The server DBMS writes these "query result data" into a so-called *query result table.* This is a temporary database table on disk in a special *data transfer format* /KDG87, GM90/. This data transfer format can then directly be used for sending the data to the workstation (next step) without any additional data conversion.

3. The result table is sent to the workstation where it is written to disk by the workstation DBMS. It is now seen as a "local database table" as part of the workstation database.

4. The result table (still in the data transfer format) can now be processed by the application program on the workstation. This is done via cursor-based operations. Complex objects can be transferred from the result table into the application program (and vice versa), can be modified, and can be deleted. New complex objects can be created in virtual memory (in the object cache) and can then be written into the result table on disk. Most of these operations can be performed in different *modes* with or without *set orientation* and with or without *complex object orientation.* By combination of complex object *and* set orientation, a large set of complex objects can be transferred from the result table to the application program (and vice versa) via a single DBMS call. This is a very efficient means for data exchange between the database and the application program /ES88/.

   The AIM-P API is currently available for two programming languages, PASCAL /ES88/ and APL2 /RKP90/. PASCAL type declarations, which may be embedded in an application program, can automatically be derived from AIM-P database type definitions /Da88/. These type declarations and the related PASCAL program variables are the program counterpart for AIM-P database objects of any size and complexity.

5. If the contents of the result table on the workstation has been changed by the application program (via insertions, updates, or deletions), these changes must finally be propagated back into the server database. This is done at check-in time. Change propagation is performed on a "per complex object basis", i.e. the changed parts (only the changed parts!) of a complex object are sent back to the server for materialization in the server database. In case of an object deletion, this means that just a "delete command" is sent back without any related data. In case of an object insertion, an insert command *and* the new data must be sent to the server. The more interesting - and also more challenging - case is a mixture of insertions, updates, and deletions within *one* complex object. Specific mechanisms like *delta propagation* (propagate only the changed data back to the server) and *multi-level flagging* (set flags to simplify change detection in a complex object) are used to minimize the amount of data which must be sent back to the server *and* to make change detection and materialization on the server an easy task /KDG87/.

As a starting point for system usage and evaluation, the AIM-P workstation-server architecture has been implemented on a single mainframe computer in a VM/CMS operating system environment. Workstation and server are virtual machines on the same computer. This is of course not yet the final workstation-server scenario. Porting to an AIX environment on RS/6000 hardware is currently under way. The ultimate goal will be to have a workstation DBMS on RS/6000 AIX and a server DBMS on VM/CMS (both may be AIM-P components). In such a *non-homogeneous environment* with data exchange between different hardware and software platforms, with different character sets and encoding rules, etc., the problem of a *neutral and system independent data exchange format* for complex object transfer must be solved. Some conceptual work and implementation in that direction has already been done /GM90/.

3.4 Summary

Workstation-server database management seems to be a very promising approach for many "non-standard" database application areas, especially in engineering/CAD. The engineering requirements w.r.t. performance, availability, and functionality of the DBMS (data modelling, query capabilities, version management, etc.) can be met - to a large extent - by a workstation-server DBMS. Currently, most workstation-server DBMS are still in the prototype stage, and there are still many problems to be solved in the implementation of *efficient* mechanisms for workstation-server cooperation. Especially, as it was shown in Section 3.2, algorithms for a *tight cooperation* between server and workstation DBMS are rather "tricky" in design and implementation. Nevertheless, we are optimistic that workstation-server cooperation with quite good solutions for these problems will be available in many DBMS products in the early '90s.

4. Distributed Database Management

Distributed file management and data exchange based on files are well-known techniques since many years. Distributed database management also has a longer tradition than database sharing or workstation-server database management. Distributed database management intends to provide a single system image of the data even if they are located on *different computers* at *different locations*. The users need not be aware of the place where the data are actually stored within a computer network. There is a (local) DBMS on each computer within the network as part of the (global) distributed DBMS. In contrast to *database sharing*, these local DBMS do not share any data, neither on disk nor in the system buffers; in contrast to *workstation-server database management*, there is no check-out and check-in processing and there are no long locks or long transactions for engineering applications in a distributed database management system.[5]

---

[5] However, the concepts of distributed database management and workstation-server database management can be integrated into a single system (see also remarks in Section 5).

Distributed database management reflects the fact that large organizations are usually not fully centralized; rather, data and applications very often reside on different computers at different locations because of application and organizational demands. Since distributed database management is already a well-established discipline and rather well-understood in computer science[6], we will not go into the details as we did it for database sharing (Section 2) and workstation-server database management (Section 3). Rather, we will shortly explain how distributed database management evolved from distributed file management and file transfer and we will then address some of the major problems encountered in the implementation of distributed database management systems.

File transfer is probably the oldest (and still flourishing) form of "data distribution": Data are written to file at one location, and the file is sent to another location for further processing. There is usually a lot of manual interaction in such a scenario, and the user himself must take care that the right data are at the right place when they are to be processed. Besides file transfer, where a file is actually sent to another location, *remote file access* is another means to access and manipulate data from a file which is stored at another location. Such as traditional file management is the predecessor of (centralized) database management, file transfer and remote file access can be seen as predecessors of distributed database management.

An excellent overview on distributed database management is given, for instance, in /Mo86/. That paper also gives a good overview on some major distributed database management systems which are either still in the prototype stage or are already commercially available as products on the market. From /Mo86/ it becomes also quite obvious that the major problems in current distributed database management technology are *distributed query processing and optimization* and *transaction management*. Most of the products which claim to be a distributed DBMS are either weak in optimization (if *one* query affects database tables on different locations) or in transaction management (if *one* transaction affects data on different locations). However, most of these more technical problems have already been solved in research to some extent and these solutions will show up in products very soon.

5. Conclusions and Outlook

In this paper, an overview was given on three major directions of distributed and cooperative database management:

- database sharing,
- workstation-server database management,
- and distributed database management.

Since distributed database management is already quite well-known in research and practice since several years (in contrast to database sharing and workstation-server database manage-

---

[6] See /CP87, Br82, OV89, Ro80, Mo86/ and many other text books and publications on that subject.

ment), it was only shortly addressed in the paper. There are also some excellent text books on that subject /CP87, Br82, OV89/ whereas similar compendia on database sharing and workstation-server database management do not yet exist. Moreover, quite a number of distributed database management products are commercially available now whereas full-scale database sharing and workstation-server DBMS products are still rare.

All these approaches to distributed and cooperative database management will find their market in the future, and there will also be *combinations* of some of these concepts: The server DBMS in a workstation-server environment, for instance, may be a distributed one or a database sharing system, and each node of a distributed DBMS may be a database sharing system again.

In order not to "overload" the paper, we decided to concentrate on these three directions of distributed and cooperative database management; some other aspects of distributed and cooperative database management could therefore not be addressed. One such aspect, for instance, deals with *remote database access (RDA)* /ECMA86/. In RDA a protocol is defined that enables application programs to access data of a remote database in a system independent way. More details about RDA and other approaches are given in the literature.

Acknowledgements

Some of the figures in Section 3 were taken from previous work done by P. Dadam.

References

BHR90 V. Bohn, T. Härder, E. Rahm: Extended Memory Support for High Performance Transaction Processing. Technical Report, Univ. of Kaiserslautern, Dept. of Computer Science, 1990

Br82 O.H. Bray: Distributed Database Management Systems. Lexington Books, D.C. Heath and Company, 1982

CP87 S. Ceri, G. Pelagatti: Distributed Databases - Principles and Systems. McGraw-Hill, 1987

Da86 P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, G. Walch: A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington, D.C., 1986, pp. 356-367

Da88 P. Dadam, K. Küspert, N. Südkamp, R. Erbe, V. Linnemann, P. Pistor, G. Walch: Managing Complex Objects in R^2D^2. Proc. HECTOR Congress, Vol. II: Basic Projects, Karlsruhe, 1988, Springer-Verlag, pp. 304-331

De86 U. Deppisch, J. Günauer, K. Küspert, V. Obermeit, G. Walch: Considerations on Database Cooperation between Server and Workstations (in German). Proc. GI Annual Conf., Berlin, 1986, Springer-Verlag, Informatik-Fachberichte 126, pp. 565-580

DL89 P. Dadam, V. Linnemann: Advanced Information Management (AIM): Advanced Database Technology for Integrated Applications. IBM Systems Journal, Vol. 28, No. 4, 1989, pp. 661-681

DO87 U. Deppisch, V. Obermeit: Tight Database Cooperation in a Server Workstation Environment. Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin, 1987

ECMA86 ECMA: Remote Database Access, Second Working Draft for a Standard, 1986

ES88 R. Erbe, N. Südkamp: An Application Program Interface for a Complex Object Database. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988, pp. 211-226

GM90 J. Günauer, W. Manus: Exchange of Complex Data Objects in a Heterogeneous Workstation-Server Environment (in German). IBM Heidelberg Scientific Center, 1990

HR83 T. Härder, A. Reuter: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, Vol. 15, No. 4, 1983, pp. 287-317

KDG87 K. Küspert, P. Dadam, J. Günauer: Cooperative Object Buffer Management in the Advanced Information Management Prototype. Proc. 13th Int. Conf. on VLDB, Brighton, U.K., 1987, pp. 483-492

Ke89 A. Kemper, M. Wallrath, M. Dürr, K. Küspert, V. Linnemann: An Object Cache Interface for Complex Object Engineering Databases. Technical Report TR 89.03.005, IBM Heidelberg Scientific Center, 1989

KG89 K. Küspert, J. Günauer: Workstation-Server Database Systems for Engineering Applications: Requirements, Problems, and Solutions (in German). Proc. GI Annual Conf., Munich, 1989, Springer-Verlag, Informatik-Fachberichte 222, pp. 274-286

KLS86 N.P. Kronenberg, H.M. Levy, W.D. Strecker: VAX Clusters: A Closely Coupled Distributed System. ACM Transactions on Computer Systems, Vol. 4, No. 2, 1986, pp. 130-146

Li88 V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, M. Wallrath: Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. Proc. 14th Int. Conf. on VLDB, Los Angeles, Cal., 1988, pp. 294-305

Mo86 C. Mohan: Recent and Future Trends in Distributed Database Management. In: New Directions for Database Systems (G. Ariav, J. Clifford, eds.), Ablex Publishing Corporation, 1986, pp. 35-50

OV89 T. Ozsu, P. Valduriez: Principles of Distributed Database Systems. Prentice-Hall, 1989

PA86 P. Pistor, F. Andersen: Designing a Generalized NF^2 Data Model with an SQL-Type Language Interface. Proc. 12th Int. Conf. on VLDB, Kyoto, 1986, pp. 278-288

Pi87 P. Pistor: The Advanced Information Management Prototype: Architecture and Language Interface Overview. Proc. 3. Journees Bases de Donnees Avancees, Port-Camargue, France, 1987, pp. 1-20

PT85 P. Pistor, R. Traunmüller: A Database Language for Sets, Lists, and Tables. Technical Report TR 85.10.004, IBM Heidelberg Scientific Center, 1985

Ra86 E. Rahm: Primary Copy Synchronization for DB Sharing. Information Systems, Vol. 11, No. 4, 1986, pp. 275-286

Ra88a E. Rahm: Concurrency Control in Multiprocessor Database Systems: Concepts, Implementation, and Quantitative Evaluation. Springer-Verlag, Informatik-Fachberichte 186, 1988

Ra88b E. Rahm: Design and Evaluation of Concurrency and Coherency Control Techniques for Database Sharing Systems. Technical Report 182/88, Univ. of Kaiserslautern, Dept. of Computer Science, 1988

Ra89a E. Rahm: A Framework for Workload Allocation in Distributed Transaction Systems. Technical Report (ZRI-Bericht 13/89), Univ. of Kaiserslautern, Dept. of Computer Science, 1989

Ra89b E. Rahm: Recovery Concepts for Data Sharing Systems. Technical Report (ZRI-Bericht 14/89), Univ. of Kaiserslautern, Dept. of Computer Science, 1989

Re81 A. Reuter: Database Recovery (in German). Carl Hanser Verlag, 1981

RKP90 M. Rösner, K. Küspert, P. Pistor: An APL2 Programming Interface for a Database System Supporting Extended NF^2 Relations. IBM Heidelberg Scientific Center, 1990

Ro80 J.B. Rothnie: Introduction to a System for Distributed Databases. ACM Transactions on Database Systems, Vol. 5, No. 1, 1980, pp. 1-17

Sc87 T.W. Scrutchin, Jr.: TPF: Performance, Capacity, Availability. Proc. IEEE Spring CompCon, 1987, pp. 158-160

Sh85 K. Shoens et al.: The Amoeba Project. Proc. IEEE Spring CompCon, 1985, pp. 102-105

SUW82 J. Strickland, P. Uhrowczik, V. Watts: IMS/VS: An Evolving System. IBM Systems Journal, Vol. 21, No. 4, 1982, pp. 490-510

TPF88 Transaction Processing Facility, Version 2 (TPF2). General Information Manual, Release 4.0, IBM Order No. GH20-7450, 1988

Tr83 I. Traiger: Trends in Systems Aspects of Database Management. Proc. 2nd Int. Conf. on Databases, 1983, pp. 1-20

WIH83 J.C. West, M.A. Isman, S.G. Hannaford: PERPOS Fault-Tolerant Transaction Processing. Proc. 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems, 1983, pp. 189-194