SFB-Bericht 26/88, SFB 124, University of Kaiserslautern, 1988

Transaction Support in PRIMA

H.-P. Christmann, M. Profit
University Kaiserslautern

Abstract

Advanced applications in technology, science and office automation require complex data management functions which cannot be satisfied by conventional database systems (DBS). Therefore non-standard DBS (NDBS) are investigated. New applications and NDBS require proper transaction concepts, since the *flat* transactions of conventional DBS are not sufficient, due to the long duration of transactions in CAD/CAM, office automation, etc. Additionally we have to deal with *distributed* transactions, because the whole NDBS is distributed across several servers and workstations. Abstraction and layering in the NDBS lead straight to nested transactions.

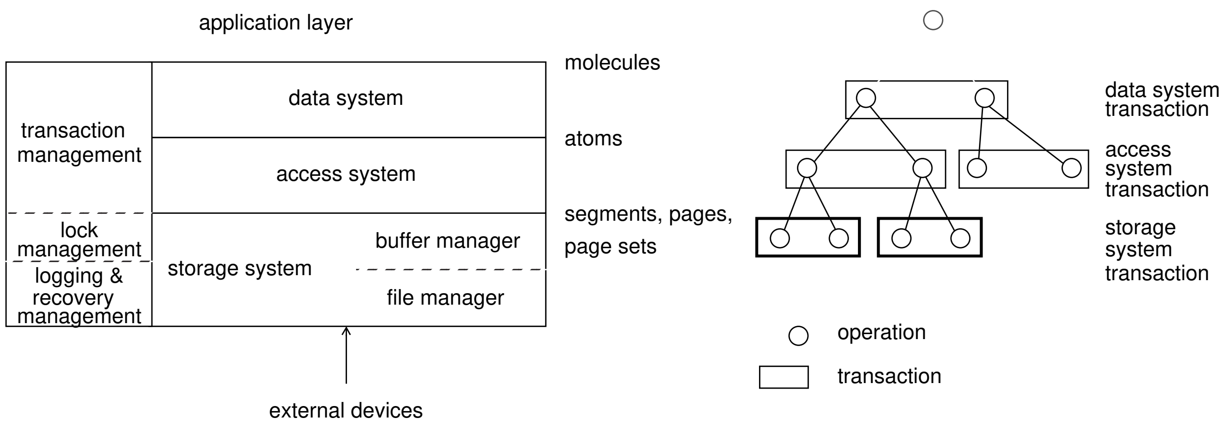
This paper concentrates on management of distributed nested transactions in PRIMA, a prototype of a DBS-kernel, and introduces synchronization, logging and recovery at the lowest levels of PRIMA. A solution for the buffer invalidation problem and synchronization and logging/recovery for FPA management are introduced. Implementation aspects are discussed with respect to PRIMA.

1. Introduction

The new application areas for database systems in technology and science, (e.g. CAD/CAM, geographical data processing as well as knowledge representation) place demands which cannot be fulfilled by conventional database systems. For this reason, so-called non-standard database systems (NDBS) are being developed which are more suited to these applications [HR85, KLMP84].

In such a system the database kernel architecture [HR85] is a key element, whereby the NDBS consists of an application specific part (application layer) and a neutral application database kernel. PRIMA (prototype implementation of the molecule-atom data model) represents such a database kernel system. PRIMA has several layers, providing access to the database at different levels of abstraction (fig. 1):

- The *data system* [Sch88] makes the molecule-atom data model available at the external interface (MAD [Mi88]). MAD is an extension of the relational data model and allows the construction of complex structures (molecules), as a set of tuples (atoms) with connection structures. Molecules may be selected, inserted, modified and deleted. Molecules are generated dynamically with each data system call and they may overlap with other molecules (non-disjoint data structures).



a) static mapping b) dynamic flow of control

Figure 1: PRIMA architecture

- The *access system* [Si88a] provides an atom oriented interface, similar to the research storage system (RSS, [As81]) of system R [As76]. The operations are read (directly and along scans), insertion, modification and deletion of atoms.
- The *storage system* [Si88b] accomplishes a database (DB) abstraction in the form of an "infinite" linear address space. This address space is divided into pages, which may be read and modified.

PRIMA is intended to run on a loosely coupled multiprocessor system in order to utilize parallel processing capabilities within application operations. Therefore, a powerful processing concept for dynamically controlling the execution of distributed actions is required [HHM86, HSS88] as well as the availability of appropriate hardware resources. Fig. 2 outlines the allocation of the PRIMA system to a server complex.

The multi-layered NDBS architecture exhibits well-defined interfaces which directly facilitate the decomposition of operations and, as a consequence, support concurrent execution of such decomposed operations. Fig. 1a sketches only the static mapping of NDBS objects and operations. The dynamic processing of user operations may be explained by a tree of suboperations as illustrated in Fig. 1b. This operation tree reflects the hierarchical decomposition of a user operation according to our architectural model, that

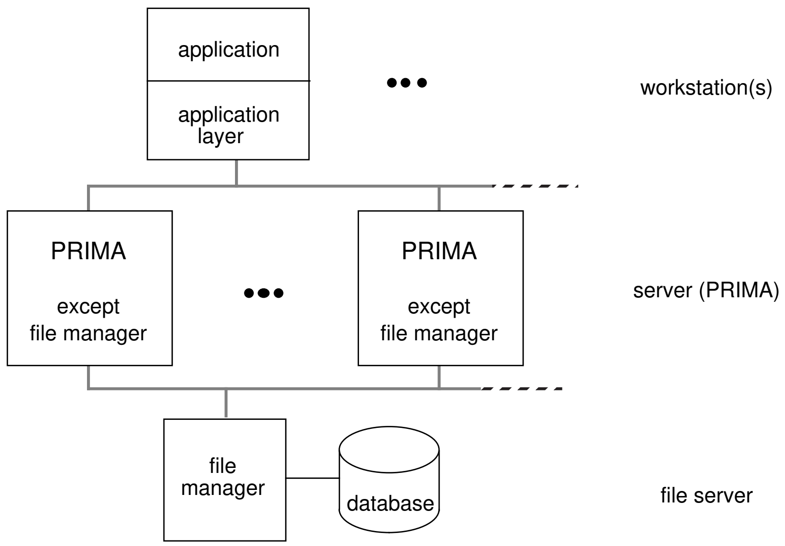


Figure 2: Hardware architecture of NDBS

is, the operations call services from the underlying layer which are decomposed, in turn, into more elementary operations.

An important design decision is concerned with dynamic execution control in an NDBS. It was often argued in the literature [KLMP84, WS84] that the conventional transaction concept would not provide a satisfactory solution for various reasons. In our case, it seems quite natural to refine the transaction concept and to adapt it to the processing structures to be controlled. Hence, the dynamic control flow of user operations can be conveniently represented as a tree of nested transactions (Fig. 1b). The concept of nested transactions [Mo81] enables the encapsulation of suboperations thereby providing isolation among concurrent suboperations and failure confinement within a nested subtransaction (isolated recovery). Only the root-level transaction is explicitly specified by transaction boundaries and visible at the application interface.

In this paper, we discuss a number of important aspects related to nested transaction management. Especially, we introduce the mechanisms for transaction support as implemented in PRIMA, that is,

- the management of the transaction trees as control information for distributed nested transactions
- synchronization support for page level objects which may be distributed across multiple database buffers
- logging and recovery algorithms to achieve atomicity and durability for low-level transactions.

Synchronization as well as logging and recovery are concerned with physical objects maintained (potentially as copies) in multiple buffers of the server complex. They provide a safe implementation of the transaction at the lowest level (so-called storage system transactions). Furthermore, they form a conceptional level upon which higher-level transaction may be built up, thereby abstracting from many physical aspects such as pages and their locations. Hence, logical (more abstract) objects and more flexible algorithms may be used to implement kind of "semantic" transaction management. For example the use of synchronization algorithms adjusted to the higher-level objects and their related operations may increase the degree of concurrent operations within the system.

The following chapter introduces the transaction management in PRIMA. The third chapter deals with the buffer invalidation problem, page level synchronization and operations for free place administration (FPA). Chapter 4 presents the logging and recovery mechanism for pages and FPA. Chapter 5 shows the system environment of transaction, lock, logging and recovery management. The paper closes with a short summary and an outlook as to future work.

2. Transaction Management in PRIMA

One point in which PRIMA differs from conventional database systems is the underlying transaction model. In conventional DBS, transactions are sequential streams of operations [Gr81]. These transactions are atomic (all or nothing): They receive the DB in a consistent state, they are carried out in isolation from one another, and the results obtained by the transaction are persistent and leave the database in a consistent state. This all is known as the ACID-principle of transactions [HR83]. Atomicity, isolation, and persistence have to be assured by suitable synchronization and recovery mechanisms.

In PRIMA, on the other hand, the concept of *nested* transactions as introduced in [Mo81] is used. This transaction model shows characteristics which make it look promising for use in NDBS.

- A transaction may contain subtransactions (STA) which, in turn, may also be composed of subtransactions. This leads to a hierarchy of nested transactions.

  The root transaction which is not enclosed in any other transaction is called the top-level transaction (TL-TA). Transactions having STA are parent transactions, and their STA are child transactions. Ancestors and descendants are all transactions reached by using the parent or child relation recursively. Children of the same parent are so-called siblings. Transactions without children are called leaves or leaf transactions. The TL-TA and their descendants form a transaction tree, where nodes represent transactions and edges the parent child relations.

- Subtransactions may be processed concurrently within a transaction, since they are isolated from one another.
- In the event of a failure during execution of one of the partial tasks, there is no influence on any of the other transactions of the transaction hierarchy. Since atomicity is required for the subtransactions, it is guaranteed by corresponding recovery mechanisms that no effects of the faulty transaction remain.
- Therefore, using nested TA in a multiprocessor system has the advantage that a system failure of one processor does not affect the whole transaction tree. Only the STA on that processor are aborted. This abort is the image the parents of this STA get from the system failure. In addition, this supports a modular structure of the whole system.

In the PRIMA architecture the TL-TA resides at the workstation. Data system and access system operations are subtransactions. Additionally, nesting within the layers is possible, but not yet used.

To guarantee the transaction characteristics transaction management, lock management and logging and recovery management are required. Transaction management is responsible for nested transaction cooperation (managing transaction trees), generation of transaction identifiers (TA-ID) and the termination of transactions (Commit, Abort). It uses the lock management to synchronize access to shared data and the logging and recovery management to guarantee atomicity and durability of the modifications (fig. 3).

Open Nested Transactions

In many conventional database systems, locking and logging at the page level is regarded satisfactory and sufficient, since the transaction characteristics can be effectively guaranteed by synchronization and recovery. This is because the transactions are only short lived and access only small quantities of data, so that conflicts between competing transactions only seldomly occur, or alternatively are quickly finished.

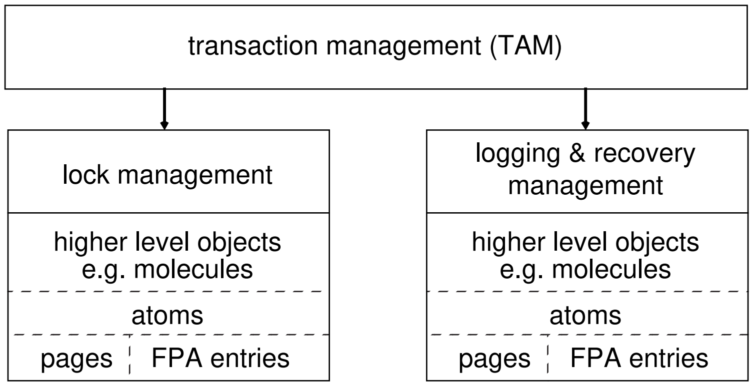


Figure 3: Architecture of transaction management

In NDBS locking and logging only at this level is inappropriate, because it may lead to unnecessary blocking of concurrent transactions due to inappropriate locking granules for a long time; furthermore, a high amount of log information has to be saved over long periods of time. To avoid this drawback, multiple granules of locking and logging may be used in PRIMA; hence, it is possible to adjust transaction management to the hierarchy of objects embodied by the multi-level architecture, that is, the concept of open nested transactions [Li84, WS84, We86] is added to the PRIMA transaction concept.

Let us consider out hierarchically nested transaction concept in more detail. The data system accepts requests from the application layer to fetch or manipulate sets of molecules (MQL-statements) which are grouped into data system transactions (fig. 1). In order to accomplish the transaction properties, appropriate measures for synchronization and logging (related to molecules) take place at this level. A data system operation is decomposed into several suboperations which may request services by the access system. All access system calls belonging to the same data system operation form a so-called access system transaction. Again, appropriate measures to guarantee the transaction properties are performed, that is, all referenced atoms are locked and all modified atoms are logged. Note, these resources are not released at the end of an access system transaction, but they are passed on to their parent transaction (inheritance of lock and logs).

An access system operation (e.g. a scan along a sequence of atoms or the insert of an atom together with the maintenance of its access paths) typically requires a number of storage system calls to perform the necessary work. All service calls of such an access system operation are enclosed by a so-called storage system transaction. It requests locks and writes log information for all pages used. Note, these resources are released at the end of each storage system transaction. This early release (w.r.t. its ancestor transaction) is possible without violating the properties of the TL-transaction, because ancestor transactions keep sufficient synchronization and recovery information on higher-level objects. Hence, we achieve kind of open nesting at the lowest transaction level.

In PRIMA most operations at the interfaces follow the "work&commit" semantic of [HR87a], that is, they form a single call transaction. Only the scans of the access system provide a conversational interface, where OPEN-SCAN starts a transaction and CLOSE-SCAN ends it (multiple call or conversational transaction).

In the following, we discuss our general transaction mechanism.

Begin Of Transaction (BOT)

The TL-TA is started in the application layer (workstation) and initiates STA at the PRIMA servers through data system calls. These STA are also nested, as said above, but will not start STA at other processors (in our current implementation). However, our transaction concept is more general and enables every STA to initiate STA (children) at other processors, in principle.

Each operation that is realized as a transaction at the lower level generates a BOT. If a transaction (parent TA) calls services from a component residing at another processor, a place holder, called agent, for the parent TA is installed. This agent represents the parent TA at the remote node and inherits the child's resources at commit.

Commit

In order to be able to guarantee the atomic termination of a transaction, a protocol has to be followed which assures, that the transaction is successfully completed on all participating processors or that all effects of an aborted transaction disappear. This can be ascertained by the two-phase-commit (2PC) protocol [Li84]. The 2PC protocol is used for every transaction (STA, TL-TA) in PRIMA and is processed among the transaction and their agents.

In our case, the transaction manager of the committing transaction acts as coordinator. The transaction managers of the corresponding agents act as subordinates. When the coordinator demands a transition to the state "prepared to commit" from the subordinates, it is assumed that the subordinates are already in the state "prepared to commit", i.e. there are no active STA. This state, which means that the data are safe in the case of a system failure and also that log information to restore the changes is available, was reached when the last STA at the agents processor committed. So phase one of 2PC may be omitted, because the result is obvious, and the coordinator can directly send "commit" messages. After this only the locks to the data and the log information is passed on upwards in the TA-tree to the parent TA or the agent of the parent.

After inheritance of locks and log information the entries for the committing TA and their agents are deleted. They are no longer necessary, since there are no active STA and the lock and log information was inherited to the parent.

Abort

If a TA is to be aborted it has to be assured that no effects of the TA survive. In this case the TAM of the processor on which the TA is running sends "abort" messages to all transaction managers on which the TA has STA which are not yet completed, or on which the TA had STA and could have engaged resources through an agent.

The agents send abort requests to the local STA. When these are aborted the inherited log information is used to compensate the modifications of committed STA. Having done this, the coordinator is informed and the entry for the agent is deleted. The aborting transaction may also have inherited log information, which is also used for compensation. At last the entry for the transaction is deleted.

In the next sections the requests at the transaction identifier and various possibilities as to their realization and the management of the TA-tree are introduced.

2.1 Realization of the Transaction Identifier

The identifier of a transaction has to assure a unique identification of the transaction. If the assignment of the transaction identifier is done centrally, and only flat transactions are present, then it is sufficient simply to number the transactions by use of a counter. The main point is the uniqueness of the identifier. Two or more transactions are not allowed to have the same ID. In PRIMA further demands are placed on the identifiers of the transactions, by the use of the concept of nested transactions and the use of a distributed system.

The TA-ID should allow a unique mapping to a TA-tree and at the same time, make the determination of the position in this tree possible. In this regard a sufficiently large fan-out has to be assured, since the number of direct descendants has to be large enough. A restriction of the tree depth is also not allowed

A decentralized assignment of TA-IDs, as in distributed DBS, requires systemwide unique identifiers. One solution is to give each processor a set of IDs, where once this supply is exhausted, more may be requested. In principle, this is still a centralized assignment of TA-IDs, however not for each individual ID, but in a blockwise manner.

A further possibility is to combine local unique TA-IDs with a processor ID in order to attain systemwide uniqueness. This approach is to be found in several systems, e.g. the distributed DBS R* [Li84].

Nested transactions [Mo82] place further demands on transaction IDs. The unique identification remains, but all transactions within the same nested TA are to be grouped together. In addition the TA-ID must reflect the position of the transaction in the transaction tree. Given a TA-ID, all ancestors and descendants must be locatable. In granting locks or checking whether the access to an object is allowed, the test as to whether one transaction is ancestor of the other must be carried out. This is done with two TA-IDs as input. To commit STAs their parent must be known in order to inherit the locks. Aborting a TA requires knowledge of all active STAs.

An obvious solution is a composition of a processor ID and an ID, which identifies the corresponding tree and the position in this tree. This can be done in three ways:

- concatenation of IDs,
- coding of relationships in IDs and
- explicit management of relationships.

Here, attention must be payed to the conditions of the surrounding system. In PRIMA these are:

- distributed transaction management, where assignment of IDs and the test for ancestors is distributed
- no restrictions for
  - depth of transaction tree
  - fan-out of nodes in TA tree (i.e. number of child TAs)
- simple test for ancestor relations
- fixed length for TA-ID

The last requirement depends on the high costs for implementing variable length IDs.

The introduced solutions will now be investigated with respect to the above restrictions. It is assumed, that a processor ID may be appended.

Creating the ID for an STA by the addition of a unique ID to the parent ID [Mo82], makes it possible to derive all ancestors from the ID of the STA (fig. 4). The ID used to expand the parent ID must be unique only among all children of the parent TA. This means that for each TA, the information as to which IDs were already used for children, is stored. This leads to high costs in the TAM, since the bookkeeping must be done for all TAs. In the distributed case, this leads to the same problems as in flat transactions. It must be assured that no two children of the same parent have the same ID. This can be done by communication (messages) among the TAMs or the IDs must be unified by adding the processor ID.

The depth of nesting in the case of concatenation is not limited, the fan-out of STAs depends on the number of IDs, distinguishable within each part of the key (fig. 4).

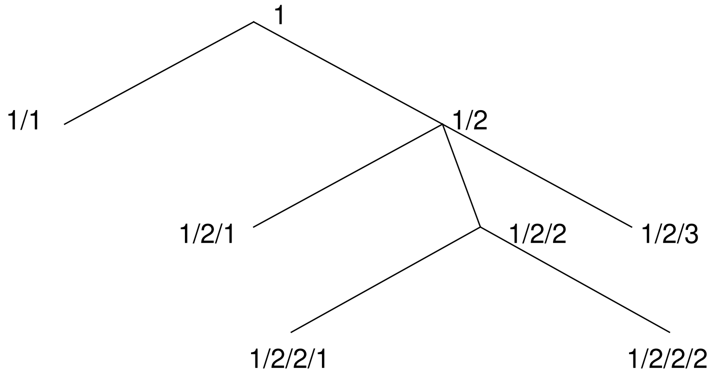


Figure 4: Concatenation of TA-IDs

The main disadvantage of this method is the variable length of the TA-IDs, which grow with the depth of the tree. This leads to problems in implementing the components that use the TA-ID (e.g. lock management, recovery, ...), as they have to use fields of variable length.

The second approach is based on coding the ancestor IDs in the new ID which is generated with an algorithm from the parent ID. One example is the use of primes (fig. 5).

Primes are used in ascending order to create the TA-IDs. The top-level TA gets 2^n as ID, where n counts the TL-TAs. Each child TA-ID is generated by multiplying the parent ID with the next available prime.

This method (there are others acting in a similar manner) allows a simple test for the ancestor relation, by division of the two IDs. If one is divisor of the other, then it is also an ancestor. The number of levels and the fan-out of the individual nodes are not limited. The main disadvantage is the extreme growth of the TA-IDs. With just a few levels and low fan-out they already exceed the range of representable numbers. So even here, variable length keys have to be used and additionally the operations to divide and multiply these numbers have to be implemented. The decentralized case leads to problems in determination of the next prime.

The third method is based on an explicit management of the tree by the TAM. Each TA gets a unique ID with fixed length (fig. 6). To allow a decentralized assignment of IDs, the incremental counted ID is expanded by a processor ID. To decide quickly between different TA trees a systemwide unique tree ID is added. The test of the ancestor relation is carried out in two steps. First of all, the membership to the same nested TA is tested. This test can be easily done everywhere, since the information (tree-ID) is contained in the two TA-IDs. If both TAs belong to the same nested TA, the TAM has to be asked for the

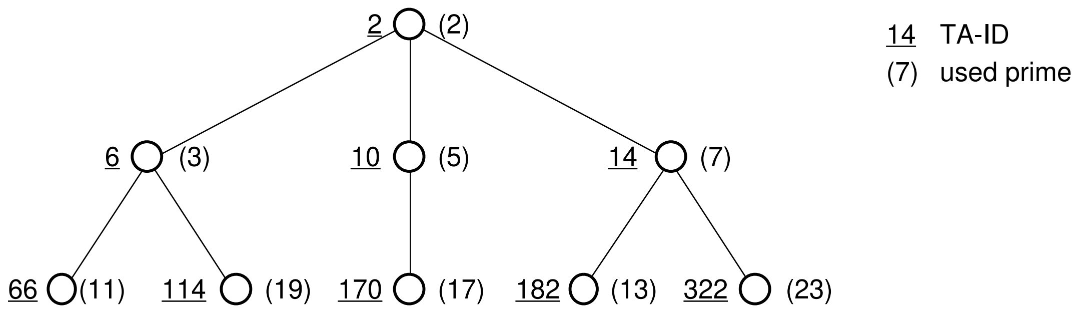


Figure 5: TA-IDs by using primes

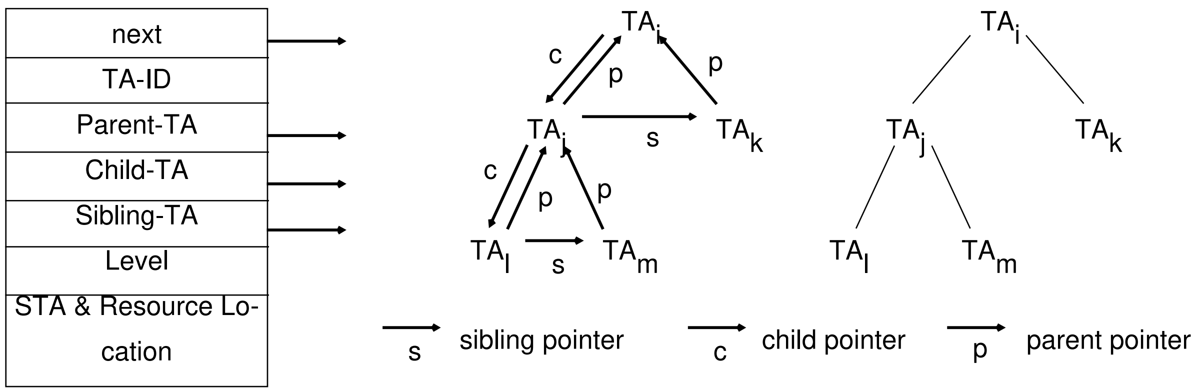


Figure 7: Transaction control block

ancestor relation, since the necessary information is only to be found there. If not, the call of the TAM can be spared.

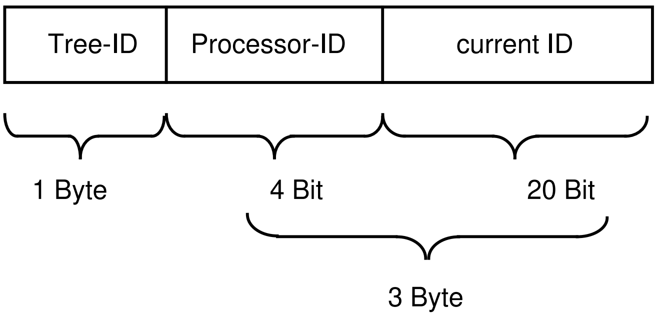


Figure 6: TA-ID with fixed length

In explicit tree management the fan-out of nodes is only limited by the number of possible IDs. Coding the TA-ID in 4 Byte, as shown in fig. 6, there are 256 TL-TAs and 16 processors in maximum, with 2^{20} STA per processor and TL-TA.

The separate management of trees allows TA-IDs with fixed length, where the length no longer depends on the position in the tree.

2.2 Management of the TA-Tree

In this section, the way in which the TA-trees are managed is introduced. The nested transaction may be distributed over several processors, where a test of the ancestor relation between two transactions, is required at each processor. The generation of TA-IDs is also necessary. First of all, the data structure for managing the tree is introduced (fig. 7).

Each TA has an entry, which contains the IDs of the transaction and its parent. This already allows the test to decide whether one TA is parent of another. When recursively used the test can decide whether the TA is an ancestor of the resp. TA. Additionally, there is the level of the TA in the tree. This information is used while testing of ancestors to stop the search as early as possible (see operation TEST). The children of a TA are referenced by pointers to the first child and to siblings. There is no order among siblings, but it is certain that all are reached. This information supports the abort processing of a TA, since all related STAs must also be aborted.

Access to the entries uses the TA-ID, which allows the support by an index structure. This can be realized using a hash-structure and leads to an additional pointer for hash overflows.

Processors running child TA or holding resources (e.g. locks), are stored in a bit list (STA & Resource Location). The TAMs of these processors are involved in termination of the TA (Commit, Abort). The processor information allows the reduction of the tree of STAs processed in commit, to one level. Abort processing is limited by the levels with active STA. Fig. 8 shows a TA-tree (a) with its distribution to processors (b). When an STA wants to commit (c), all STA have already committed and the processors are known to the actual STA. Thus the messages can be sent in one step to all relevant TAM. In case of abort (d) there might be active STA which are not yet known to the aborting STA. So all known TAM (3-7) are informed, and they recursively inform (abort) their STA (8,9). The arrows represent entries in the "STA & Resource Location"-field.

Operations on transactions are always given to the local TAM, which then eventually distributes the request to the responsible TAM. At some time information located at TAMs on other processors is needed. Communication among TAMs occurs also in case of EOT (Commit, Abort), when the TA has there STA or resources.

Operations provided by the TAM are described in the following:

BOT (Begin of transaction)

BOT creates a new STA for a given parent TA or a new TL-TA and returns the assigned TA-ID. For a TL-TA the entry is generated immediately and the ID is built up from the three pieces shown in section 2.1.

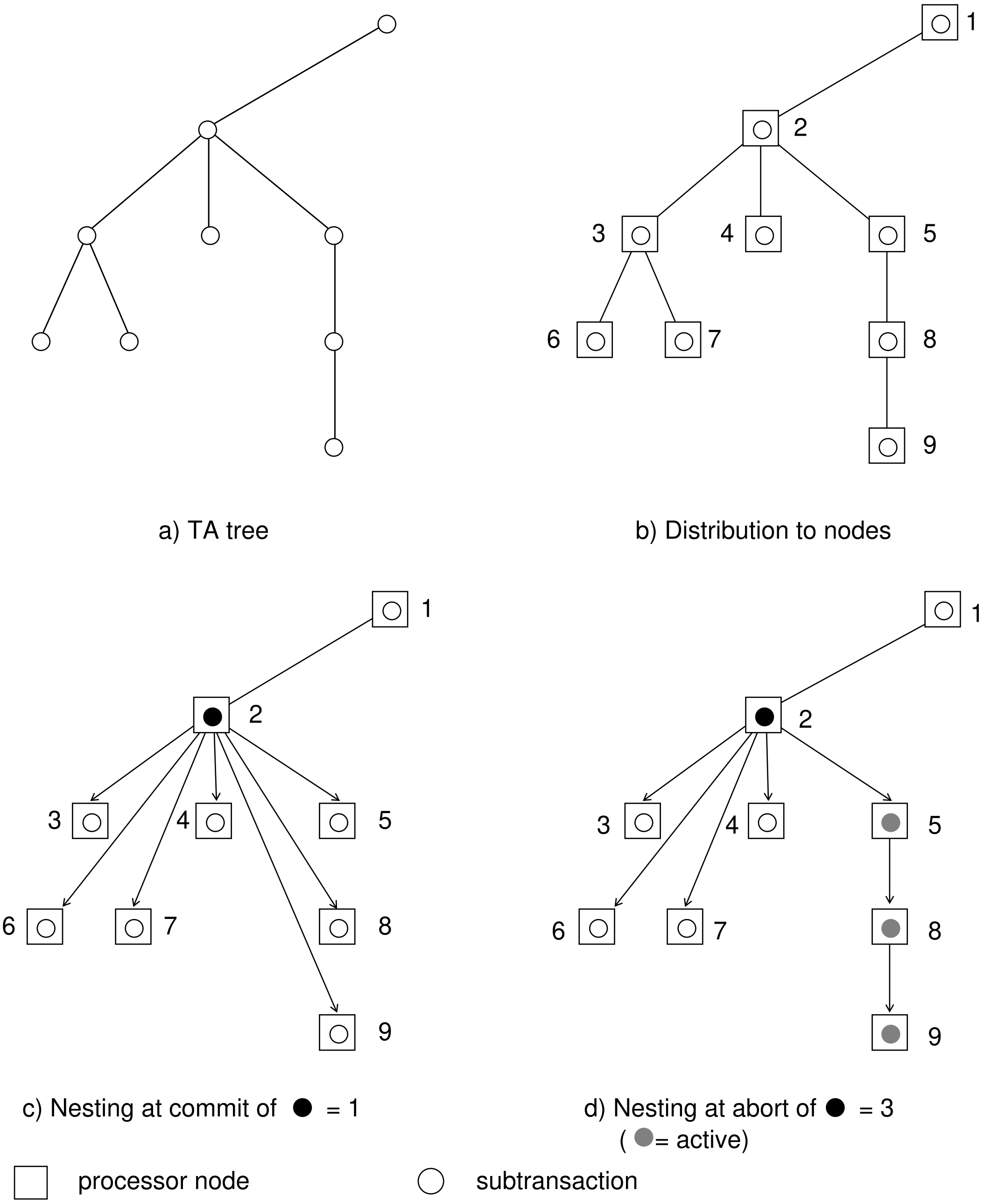


Figure 8: Modification of TA structure during EOT-processing

STAs are given the correct tree identification, which is derived from the parent ID. The current number is incremented and assigned and in addition the current processor is used for processor id. The field showing the location of STAs is initialized with zero, since no STAs are present.

For each STA of a TAM all ancestors must be known. If not, this information is requested from the home of the parent TA. From this request the TAM of the parent learns that the parent TA has a new STA at another processor, which then is marked in the "STA & Resource Location" list.

The ancestors are inserted as agents into the TAM. Then the new TA is inserted and connected to parent and siblings. In fig. 9, transaction T_k becomes a new child TA of T_i and therefore a sibling of T_j. Since no order among siblings is defined, T_k is inserted before T_j in the sibling chain.

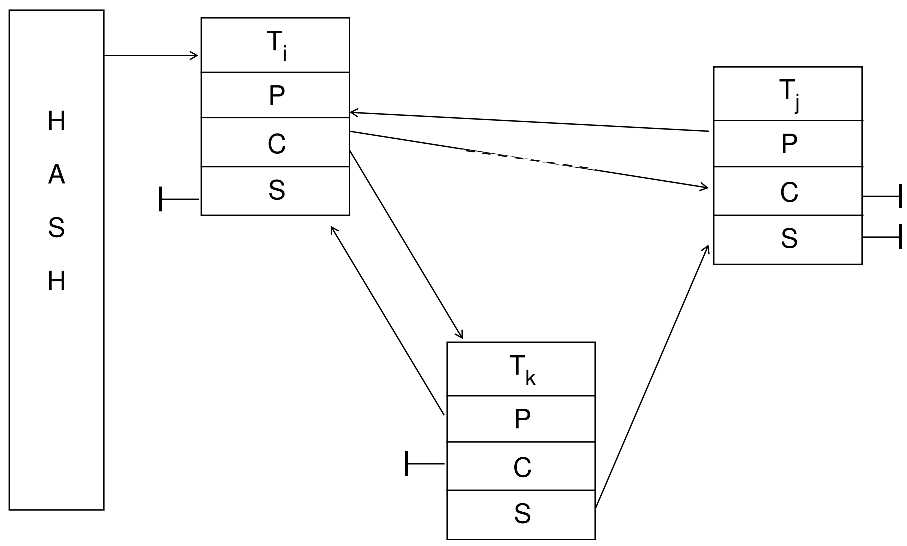


Figure 9: Insert of transaction T_k into the TA tree

COMMIT

COMMIT of a transaction is only possible, when all descendants have committed. A prepare message is sent to all TAM where resources are held. Since agents are "prepared to commit" a positive reply is certain. After commit is assured the TA and its agents inherit the resources to the parent TA or the local agents of it. In fig. 10, T2 first initiates a 2PC with A2, its agent at processor P2, which leads to a positive result. Then T1 inherits the locks L1, L2 from T2, additionally its agent A1 at P2 inherits L3 from A2. After this upward inheritance the locks are retained by the transaction [HR87b]. In fig. 8c, there are more agents for the committing TA. All are reached by one broad/multicast. No further agents which are unknown to the committing TA have to be informed. This works because the STA tells its parent during commit the locations of all resources it knows. After commit of T2, T1 inherits all resources and gets knowledge that there are agents and resources at processors 2 to 9, by sending the "STA & Resource Location" list. When the TL-TA (T1) commits, all resources such as locks and log information are released.

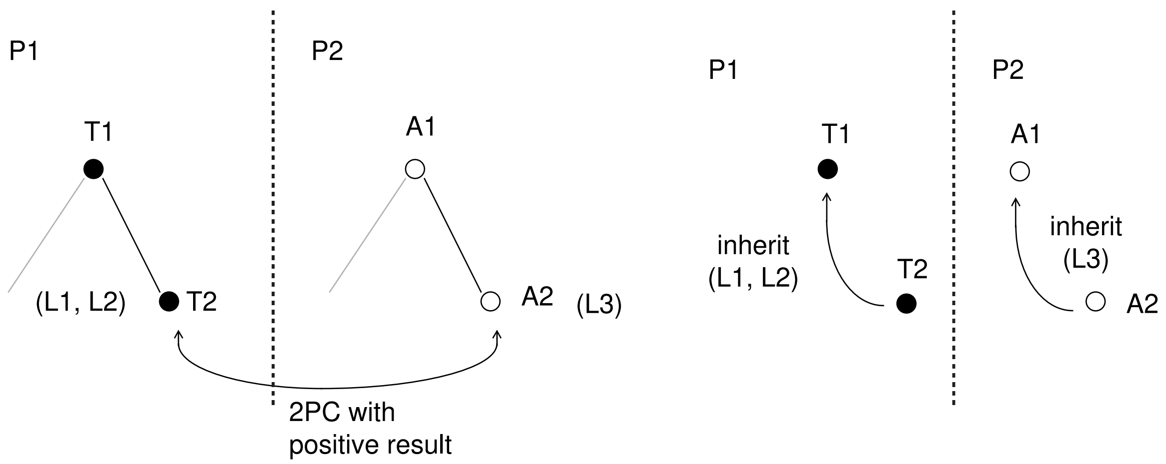


Figure 10: Commit of STA

ABORT

Abort messages are sent to the locally active STA and to the agents at other TAM, which also inform the locally active STA (A(1) messages, fig. 10). This is repeated for the informed STA (T2, T3, T4) and leads

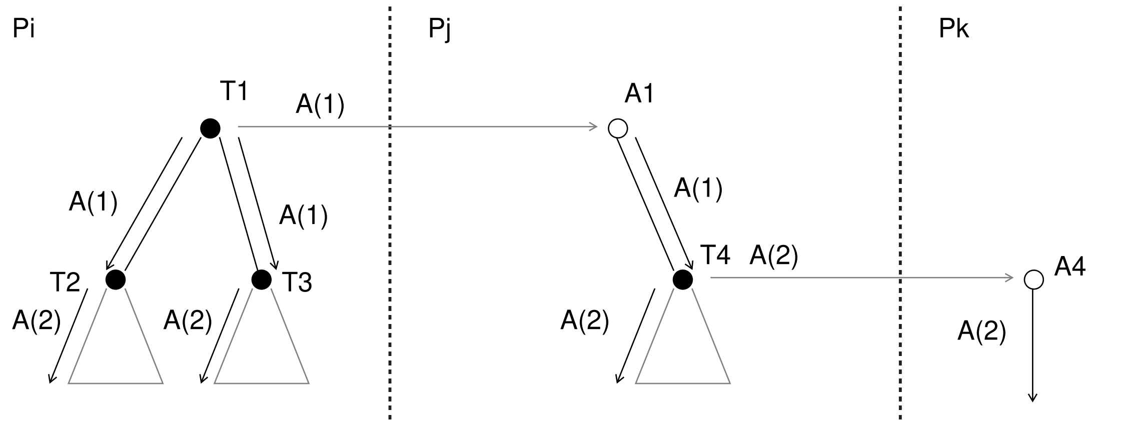


Figure 11: Abort of a transaction across nodes and down the tree

to the A(2) abort messages, and so on. When the active STA are aborted an abort is sent to the recovery manager, to undo the effects of committed STA, and to the lock manager in order to release the locks inherited from STA. In fig. 8d, T2 sends abort messages to processors 3 through 7. Processor 5 runs an active STA which has to inform further STA on processor 8 and which informs STA on processor 9. Here we have three serial messages with responses until the abort from T2 is completed. This was omitted in commit, which needs only one message.

TEST

TEST is used to ascertain the relation between two STA in the same TA tree. This cannot be seen directly from TA-IDs. When the TAs belong to different trees none can be the ancestor of the other. The operation does not only test whether one is an ancestor, it also gives the first common ancestor of the two TAs.

For each TA the level in the tree is given (fig. 6), so the search starts at the "deeper" TA. The ancestors are read level by level. When the level of the second TA is reached, a check is made as to whether it is the current ancestor. If not, the search runs in parallel for the common ancestor.

3. Synchronization

In this chapter, we will concentrate on synchronization in the storage system using page locks. The distributed processing in the loosely coupled architecture, where each processor has its own system buffer maintained by a corresponding buffer manager, leads to the storage of multiple copies and thus to an invalidation problem. The database pages are used at several processors (fig. 12), so we have to synchronize accesses from different processors to the same page (distributed synchronization).

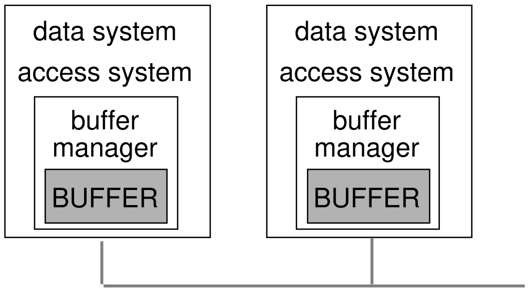


Figure 12: Buffer location in a loosely coupled system

3.1 The Buffer Invalidation Problem

The invalidation problem is known from DB-sharing systems [Ra86]. It arises when an object (e.g. a database page) is stored in multiple copies, which can be modified independently of each other. Additionally, each time the valid copy has to be processed. The change of one copy invalidates all other copies. Each of the PRIMA kernels contains a system buffer in which the pages of the DB are provided by the storage system, consisting of buffer manager [Si88a] and file manager. If a page is used in multiple kernels, then it is provided in the corresponding buffers. Changes can be made in every buffer, making the other copies invalid. When a page is requested by a FIX-PAGE operation in one of the buffer managers, the storage system has to answer two questions:

- is the available copy still valid, and
- if not, where is a valid copy?

The first question avoids the reference to an old copy. Related problems are discussed in [YYF85, Ta76], which propose several solutions. One solution demands that the modifying buffer manager sends notifications of the change to all others. If it does not know which buffer managers hold a copy, it has to notify all of them. All have to check whether they are affected. This test is not necessary if the page is not in the buffer or is no longer being accessed.

In [Ta76] centralized information about the state of the data in the buffers is used in order to inform only those buffer managers whose information is invalidated. The buffer managers notify a central "invalidation manager" of the changes, which then informs the buffer managers having a copy.

PRIMA utilizes this strategy in a slightly modified version: Central information reflecting the state in the buffers is used, but no notification of affected buffer managers takes place. The buffer managers check the validity, by reading this invalidation information, of the copies before they use them. The central information contains a list for each page, in which its state at every buffer manager is recorded (This list is called invalidation vector [Ra86, CS85]).

Pages, that are only read, may be valid in several buffers. After modification, the page is only valid in one buffer and the entries in the invalidation vector for other buffer managers are set to invalid (fig. 13).

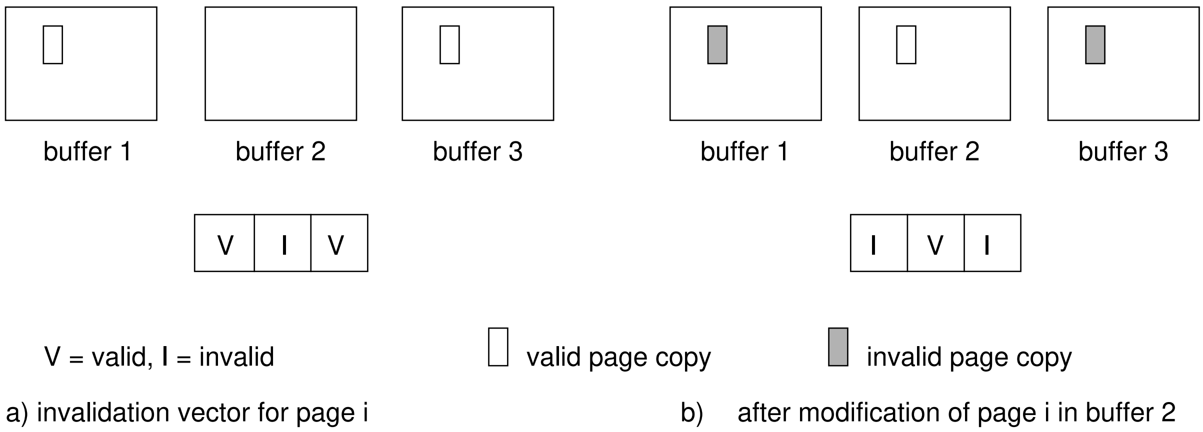


Figure 13: Invalidation vector at different buffer managers

Reading and updating the invalidation vector leads to a high message flow between the processes. Therefore, the realization of this idea is only tolerable if the cost of the messages can be reduced substantially.

Thus, the solution of the invalidation problem is integrated into lock management. Now the messages to test the state of the pages are omitted, since before the page is accessed, the corresponding lock is requested and the state information is combined with the message for the granted lock.

This procedure allows the detection of whether a page in a buffer is invalid or not. If it is invalid a valid copy must be provided by the storage system. The storage system must find the location of a valid copy (see question 2 above). This location, however, depends on the used commit-processing strategy. If the pages are directly written to disk (FORCE), the valid copy is always on disk. In the case of delayed write (NOFORCE) the valid page may be on disk, but it is also possible that the modifying buffer manager still holds the page in its buffer. Since PRIMA currently uses FORCE, a valid page is always on disk.

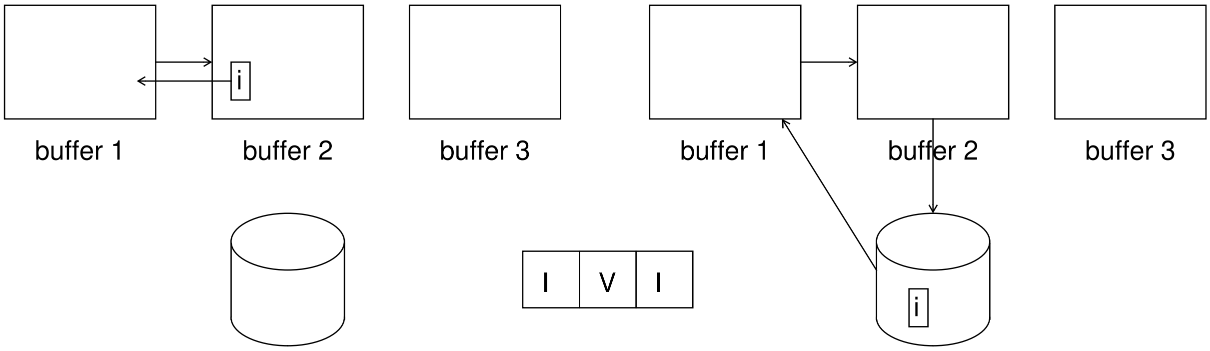
To discuss further optimization the actions in the case of NOFORCE are now briefly introduced. Here, the buffer manager referencing the page requires the following information:

- the validity of a present page, if it exists, and
- the place of a valid copy.

The second information always points to the buffer manager creating the valid copy. If it is no longer there due to replacement, then the DB contains the page and it can read there. If the page is in the buffer but not yet written to disk, it has to be decided when this shall take place. Normal replacement leads to the lost update problem, in the case that the second buffer manager modifies the page and writes it to disk earlier. To solve this problem two procedures are distinguished

- The first solution might be to delay providing the page until it is written to DB.
- The second solution is to transfer the page as well as the task to write the page to the requesting buffer manager. If the page is only read, it will be written on its replacement. In the case of successful modification only the new copy is written to disk. Unsuccessful modification then requests the write of the page's former state.

In PRIMA the buffer manager receives information as to where valid copies might be. This information is used to avoid access to disk. If one of the buffer managers has a valid copy, the page will be requested there. This avoids one access to disk (fig. 14a). However, in the case that the page has already been



a) page i is sent by buffer manager 2

b) page i is read from disk

Figure 14: Request of valid copies

replaced the corresponding buffer manager requests the page from disk (fig. 14b). In the worst case, there is an additional request to the other buffer manager compared to direct request from the disk. In the best case, the disk input is spared. In the case of changing FORCE to NOFORCE, no modifications for invalidation management and lock management are necessary.

The information regarding where a valid copy might be, is read from the invalidation vector. The lock and invalidation information are stored together in order to enable integration of invalidation detection into lock management. If there are no entries for the page, then it is available only on disk and must be read from there.

3.2 Synchronization for Storage System Transactions

Every access system operation is processed as a transaction at the storage system level and therefore called SS-TA (storage system transaction). Locks requested within one SS-TA are released at EOT, whereas locks at higher levels (entries, atoms) are retained. Synchronization at the page level [Ch87, Kr88] is done by local lock managers (LLM) at each server processor and one global lock manager (GLM), located at the file server processor, coordinating the LLM (fig. 15).

The LLMs may get an authorization from the GLM to grant read or write locks locally ("sole interest"-concept [RS84]). This procedure reduces the number of messages between GLM and LLMs. If there is a high locality of lock requests at the single servers, most lock requests may be decided by the LLM, without contact to the GLM. Another LLM requesting incompatible locks for such a page destroys locality and forces the GLM to revoke the authorization from the first LLM. The authority to grant read locks (read-

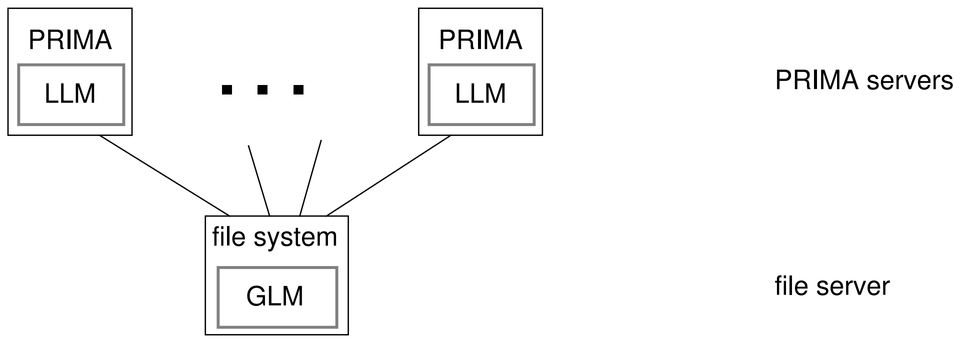


Figure 15: Lock manager structure

authority) can be given simultaneously to several LLM. Write-authority is given to one LLM only, when no read- or write-authority exist. Hence, the GLM now only manages authorizations and no longer locks.

All locks are requested at the LLM. When the LLM has the corresponding authority, the lock request is decided locally (fig. 16). Otherwise, the LLM requests authority at the GLM.

		read-authority	write-authority
requested	R	+	o
lock mode	W	-	o

+ may be granted
- must not be granted
o may only be granted, if no local conflict

Figure 16: Local lock grant with authority

To support locality of lock requests LLMs retain authorities, even if there are no longer granted locks. For this reason, incompatible authorities requested by some other LLM must be revoked by the GLM. In granting an authority, the same protection as in granting a lock is given. Accordingly, as long as the authority is available, no other buffer manager may change the page so that a given copy will be certainly valid.

The described procedure leads to the desired effect of reducing global messages while solving the buffer invalidation problem. Furthermore, the messages for lock requests are reduced as a result of the sole interest method.

For each locked page a lock control block (LCB) is installed (fig. 17). The granted lock queue and lock request queue contain transaction IDs. Lock requests have the following effects:

- When granting a lock to a transaction the corresponding TA-ID is added to the granted locks queue, which will contain only one TA-ID, when locked in write mode and may contain several TA-IDs, when locked in read mode.
- When a requested lock is incompatible with the granted lock(s) the TA-ID, together with the requested lock mode, is added to the lock request queue, which then might contain requests for read and write.

The deadlock detection problem is currently solved by an appropriate "timeout" algorithm..

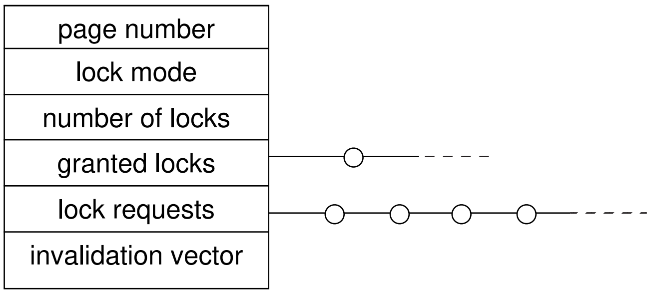


Figure 17: Lock control block with invalidation vector

The invalidation vector is updated as a result of lock requests and lock releases (fig. 18).

- If no lock control block exists and a lock for a page is requested, then one is generated and the invalidation vector is initialized with "invalid". Only the buffer manager which requested the lock receives the entry "valid".
- If the LCB exists, then locking the page changes the entry of the corresponding buffer to "valid".
- Releasing a read lock does not change the invalidation vector.
- Abort of a transaction does not change the invalidation vector of read pages, for modified pages it is set to "invalid" for the corresponding buffer manager. Commit of a transaction and release of a write lock sets the entries of all buffers to "invalid", with the exception of the modifying one which receives a "valid".

Operation on page i at processor j *)	Modifications of the invalidation vector (IV)	
read request & grant	IV[j] := valid	
write request & grant	IV[j] := valid	
release of lock by a successful operation	mode = read:	IV unchanged
	mode = write:	IV[k] = invalid, k = 1..n, k <> j
release of lock by an incomplete operation	mode = read:	IV unchanged
	mode = write:	IV[j] = invalid

*) IV is initialized with invalid when a lock control block is created

Figure 18: Modification of invalidation vector due to lock requests

3.3 Synchronization of FPA Information

The FPA (free place administration) management maintains the free place for pages inside each segment [Si88b]. The management has two levels. The header of each data page contains information regarding how much space is free in the resp. page and at which position it begins. Besides this, there are special pages in the segment (FPA pages) which contain information about the free space in each data page of the segment. The FPA page contains one entry per data page. The manipulation of this segment internal information (FPA pages) is obtained by the operations of the FPA management at the interface to the access system. The page internal information is directly modified by the access system without control of the FPA management.

The FPA management provides the operations INCREMENT-FPA, DECREMENT-FPA and MODIFY-FPA for adjustment of FPA pages to the page internal FPA information. It is assumed, when free space is requested by GET-FREE-PLACE, that the FPA pages contain up-to-date information. GET-FREE-PLACE chooses a data page and returns its number. It now has to be prevented, that this data page is used again for free place requests, since the FPA information is or will become inconsistent. Otherwise, there is no guarantee that GET-FREE-PLACE will return a data page containing enough space. The FPA page must be up-to-date when the data page is released (at UNFIX of the page or at commit or abort of the SS-TA).

There are two ways to protect the FPA entry of the data page:

- Locking the whole FPA page
- Locking the FPA entry

The first solution is very restrictive, since it blocks all other data pages whose entries are in that FPA page. Therefore, protection of the segment internal FPA information is better achieved on an entry basis. Only the entry of the corresponding data page is locked, until the two FPA informations (in FPA page and in the data page) are consistent, which happens at unlock time of the data page. Locks for the FPA page itself are immediately released after modification of the FPA entry. If a locked FPA page is found during search for free space, it may be waited for the release of this lock. In this way, an undesired behavior (e.g. unnecessary growth of segments) can be avoided to a great extent. Locks on an entry basis need logging for entries, which will be introduced in section 4.3.

The processing of the FPA management operations will now be demonstrated with a special look to locking.

GET-FREE-PLACE
A read lock is held for the FPA pages while looking for convenient data pages. The entry of the found data page is locked so that no other TA may see the free space in that data page, until the operation in the data page is completed and the FPA information is consistent. After locking the FPA entry the lock of the FPA page may be released.

MODIFY-FPA (INCREMENT-FPA, DECREMENT-FPA)
The FPA page is locked in write mode. After changing the entry and logging it, the FPA page is unfixed and unlocked. The FPA entry remains locked until the data page lock is released.

Realization of Entry Locks for FPA Entries

The basic idea of entry locking is to express the entry lock implicitly by the page lock of the corresponding data page. When a data page has a write lock, the corresponding FPA entry is also locked. This holds even in the case when the content of the page is modified without changing the length and therefore the FPA information. GET-FREE-PLACE then looks for another FPA entry, since the data page for this entry will not be available.

The lock manager provides the operation LOCK-FPA-ENTRY. In contrast to LOCK-PAGE, LOCK-FPA-ENTRY does not lead to wait situations. If an entry is locked by another TA, the lock request is not added to a wait queue, but the requestor is told that the lock is refused. The FPA management can now select a new data page which contains enough space. If all possible FPA entries have been checked and no lock has been granted, a new data page is allocated.

The data page is already locked in write mode for the TA when it is requested by a FIX-PAGE operation from the buffer manager. So processing the FIX-PAGE, it is ascertained during the resulting lock request that the page is already locked for TA and the request is answered positive.

Since no wait situations arise while looking for free space, deadlocks are avoided at this point. Pages locked by other transactions are not considered for free space requests. The test for availability of a data page prevents the waiting for locks.

4. Logging and Recovery

Besides synchronization we need logging and recovery mechanisms to support atomicity and durability of the transaction concept. These important transaction properties might be corrupted by several types of failures described in [HR83]. In this paper, we discuss logging and recovery issues for low-level transactions (SS-TA) dealing with physical pages. Recovery from two types of failures has to be supported. Transaction failures provoked by some kind of misbehavior or by an explicit abort command of the transaction or some of its ancestors imply a transaction UNDO, that is, a rollback of all modifications (and release of the corresponding locks) performed by the resp. transaction and its descendants. System failures are characterized by loss of the current DB state (DBMS and buffer data) in volatile memory such that continuation of normal DBMS processing is not possible. Hence, system failure recovery has to guarantee that the modifications of all committed (recovery) transactions (see [KLMP84]) survive and that all effects of incomplete (recovery) transactions are rolled back. It is the task of the logging component to collect and manage sufficient redundant information, which enables the recovery component to handle transaction and system failures.

Here description of logging (and recovery) is restricted to the storage system [Si88b]. Several types of objects are usable for logging, such as physical records or pages. In PRIMA, we have chosen state logging at page level. The reason for this decision was the definition of the page-oriented storage system, to which the physical logging and recovery module should be connected to gain a clear system structure. A further reason was the simple concept which leads to a less faulty implementation. A third aspect arises from the file system interface [Fr87], which is used for writing and reading the log information. It provides an operation to copy blocks directly from one file to another, instead of reading them into the log component and then writing them to the log file. These reads and writes would lead to additional messages between different processors, whereas the simple COPY-BLOCKS command is done with one message and its response.

Now logging uses the same objects (pages) as locking. Since locks at the page level are replaced by locks at the atom level when the SS-TA commits an UNDO of committed SS-TA, because of an aborted ancestor, is not possible using page log information. Because it might affect the modifications of other transactions in these pages. Thus the SS-TA can only be compensated, which requires logging at a higher level of abstraction (atoms) replacing the page level logging. Therefore, successful recovery at the lowest level (SS-TA) has to guarantee the consistency of storage structures to enable recovery at the atom level.

PRIMA is designed for a multiserver architecture (fig. 2), where each server runs an instance of PRIMA. These instances are independent with except to the global information such as synchronization and up-to-date pages. In particular, each buffer is local to its server. And hence, the logging and recovery manager is local, because SS-TA are completely executed at one processor, so the log information is not distributed. For the first approach, the recovery managers at the different sites need not communicate with each other. This approach is possible, since only one transaction at a time may access (lock) a data page in write mode and therefore request logging of this page. Recovery from a system failure is done after restart of the processor.

4.1 Log Information at Page Level

The type of log information depends on the strategy used for handling modified pages in the database buffer [HR83, Si88b]. The relevant criteria are

- update propagation, whether the modified pages of a transaction are written uninterruptedly to disk (ATOMIC or NOTATOMIC),
- page replacement strategy, which might remove modified pages of incomplete transactions (STEAL or NOSTEAL), when additional space is necessary, and
- commit processing, where we can decide between writing all modified pages to the database when a transaction commits (FORCE) or leaving them in the buffer until replacement due to space requests (NOFORCE).

Transaction-oriented ATOMIC propagation mechanisms imply mapping redundancy built in the storage structures and do not need logging and recovery mechanisms for the failure types discussed above. However, such mechanisms turn out to be very expensive during normal operation. For further discussion, we deal with NOTATOMIC which is implemented in the storage system. In the case of a STEAL policy, uncommitted data might reach the database necessitating an UNDO recovery. As UNDO information we need before images (BFIM) of the modified pages. A NOFORCE policy requests REDO recovery and log information based on after images (AFIM). The combination of page replacement and EOT-processing leads to four types of log information shown in fig. 19.

In the NOSTEAL/FORCE case we need one of the log informations due to NOTATOMIC page replacement, because a failure could occur during FORCE of the modified pages. The STEAL/NOFORCE constellation requires both AFIM and BFIM logging, although the informations are needed at different times. The BFIM has to be written before a modified, uncommitted page is written to disk, whereas AFIM log must be written during commit processing.

In PRIMA, the storage system uses NOTATOMIC/STEAL/FORCE policy. The FORCE policy has some advantages in the loosely coupled architecture. If the page is accessible, with respect to locks, the database (disk) holds the valid page. According to fig. 20, we have to log BFIMs due to STEAL policy. FORCE prevents the partial REDO and therefore does not require the AFIM-log.

	page replacement	
EOT-processing	STEAL	NOSTEAL
FORCE	BFIM	BFIM or AFIM
NOFORCE	BFIM & AFIM	AFIM

Figure 19: Necessary log information for different buffer strategies

4.2 Logging and Recovery on Page Level

The logging and recovery manager [Ho88] provides the operations

BOT	to write a BOT record to a log file,
COMMIT	to write an COMMIT record to a log file,
ABORT	to start transaction UNDO on the given transaction,
LOG-PAGE	to write the BFIM of a database page,
DELETE-LOG-PAGE	to delete the BFIM of a page from the log file and
START-CRASH-RECOVERY	to start global UNDO.

Through operations BOT, COMMIT, ABORT the log component knows about transaction boundaries. The LOG-PAGE operation is used by the storage system to dictate which page has to be logged due to modification. This operation is called when the storage system gets a FIX-PAGE operation with write option. This point is "long" before the log information is actually necessary in the case of page replacement. If the log request would come when the page has to be replaced, the storage system would have to wait for completion of logging the page and afterwards wait for writing the page to disk. Thus eventually, log information will be written even if none is necessary in the case that the modified page is not replaced. However, the early log-request in combination with the asynchronous interface avoids delays of the storage system later on, because logging is executed concurrently to normal data processing.

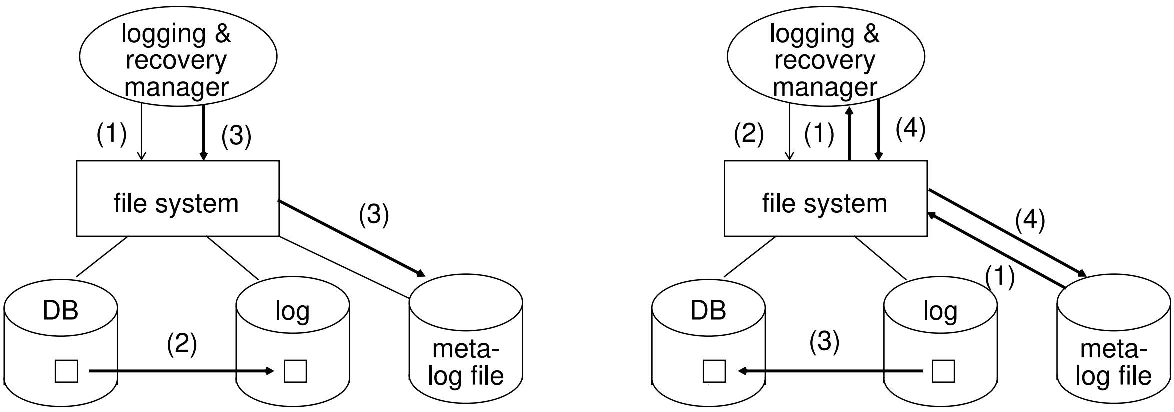
The logging is simply done by using the COPY-BLOCKS operation of the file system [Fr87], so the database block is copied to a chosen block in the log file. This pairing of blocks has to be noted somewhere. For this reason, we use a further log file (here called meta-log file), which contains records of the following type (fig. 20).

"Operation" is of type BOT, COMMIT, ABORT, or LOG-PAGE. The further fields are only necessary in the case of a LOG-PAGE operation. The entries for database block and corresponding log block consist of the file name and the block number in this file. These can be directly used as parameters for the file system. The BOT entry might be omitted, but it is written for completeness. Several of these records fit to one block, which allows the reduction of the I/O-rate.

TA-ID	operation	database block		log block	
		segment file name	block number	log file name	block number

Figure 20: Log entry in meta-log file

Logging follows the described procedure of fig. 21a, where first the COPY-BLOCKS call (1) to the file system is made, then the copy (2) is executed. At last a page with meta information (fig. 20) might be written to the meta-log file (3).



(1) COPY-BLOCKS call to the file system
(2) execution of copy to log file
(3) eventual write of log-entries to the meta-log file

a) logging

(1) read of meta information (crash recovery)
(2) COPY-BLOCKS call to the file system
(3) execution of copy to DB
(4) deletion of log entries

b) recovery (TA UNDO, crash recovery)

Figure 21: File operations for logging and recovery

An unusual operation provided is DELETE-LOG-PAGE, which removes a page from the log. This is used in two cases. First, when no modification of the data page was done, which would not be enough reason for the operation, because it only shortens the recovery procedure for the cost of an additional call to the logging and recovery manager. It is mainly used for FPA pages, for which a special lock and log processing is done (see section 4.3).

For copying the database block the log file must have the same block size as the segment which leads to one log file per block size (0.5, 1, 2, 4, 8 kbytes, fig. 22). The data written to the meta-log file are additionally held in main memory, for active transactions, to speed up transaction UNDO.

When a transaction commits, then the log information (BFIMs) may be deleted. Since this is not possible in an atomic procedure an interrupt (system failure) might occur. Therefore the COMMIT record is written to the meta-log file before the operation is acknowledged. This also speeds the commit processing for the storage system. The single log informations are deleted asynchronously. The COMMIT record in the meta-log file prevents that the modifications of a completed transaction are rolled back after a restart.

Recovery has to be done on behalf of an ABORT or after a system failure. All incomplete transactions are rolled back. A partial REDO is not necessary since all modifications were FORCEd to disk by the buffer manager. A transaction UNDO has to be processed on the database and on the buffer. The recovery in buffer is very simple, the modified page has just to be deleted in the buffer, which can be done by the buffer manager itself. The next request for the page then leads to a read from the database. The re-

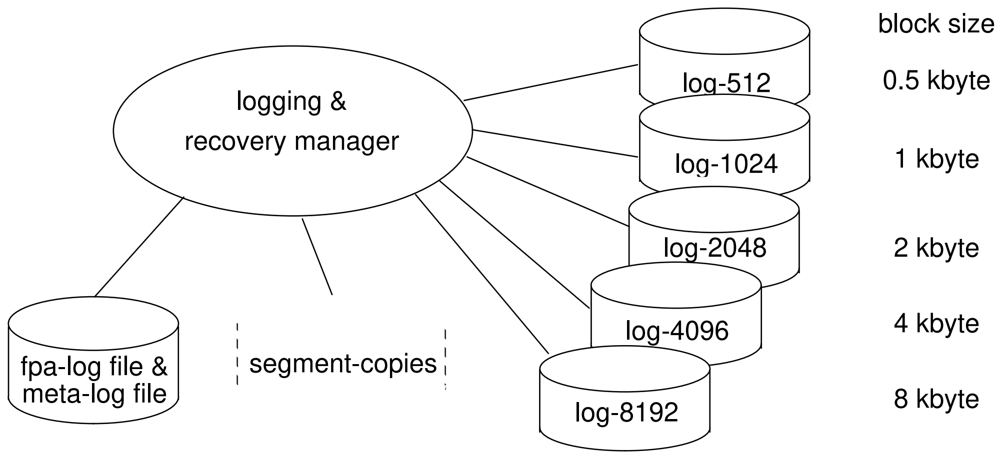


Figure 22: Files used for logging

covery for the database is done in the same way as logging (fig. 21b), by copying the logged blocks back (2,3), using the meta-information. In the ABORT case this information can be taken from main memory, where it is held. When the information has become obsolete, only a write (4) is necessary. No reads from the meta-log file or from the data-log file take place. When crash recovery has to be done, initiated by START-CRASH-RECOVERY (global UNDO [HR83]) no information is in main memory, all must be read (1) from the meta-log file. Afterwards copying back the logged blocks (2,3) takes place. After successful crash recovery all log informations are deleted and processing continues with empty log files.

4.3 FPA Logging and Recovery

In this section, the description of management of FPA pages and entries is completed. As the lock manager, logging and recovery components use FPA entries. But as in locking first FPA pages are logged and then replaced by FPA entry log information, when the operation was successful. The logging and recovery of FPA pages is managed within page logging and recovery.

FPA logging and recovery management provide the operations LOG-DELETE-SEGMENT, LOG-INIT-SEGMENT and LOG-MODIFY-FPA to save the information for the FPA operations DELETE-SEGMENT, INIT-SEGMENT and MODIFY-FPA of the FPA management. The operations INCREMENT-FPA and DECREMENT-FPA can be matched to MODIFY-FPA and therefore need no own log procedure. FPA logging writes entries to the fpa-log file (fig. 21). Fpa-log file and meta-log file are one file, to assert the correct order of recovery actions.

LOG-DELETE-SEGMENT saves the segment and its meta-information, if they are deleted by the DELETE-SEGMENT operation. Log information contains the segment description, such as segment name and segment number, page size, etc., which is written to an fpa-log file. Because a segment may be deleted, even if it still contains database information, this also has to be logged, by copying the corresponding file (fig. 21, segment-copies). So the name or ID of the segment's log file must be added to the entry in the fpa-log file. Recovery is done by initializing the segment again and restoring the data (copy back the file).

LOG-INIT-SEGMENT needs only the segment name as log information. During recovery the new segment is deleted, i.e. the file and the meta-information of the segment are deleted.

The LOG-MODIFY-FPA operation needs some further attention. An FPA log entry contains the "name" of the FPA entry, i.e. the number of the corresponding data page, and the value of the entry before the modification. Fig. 23 shows the lock and log operations of the MODIFY-FPA operation. FPA pages are only locked as long as an FPA entry is processed (1-7). The page locks are released at the end of the operation, i.e. a commit of the FPA page. After logging (2) and modifying (3) the FPA page the FPA entry lock (4) is requested. To allow recovery for the FPA entry, the page log is replaced by an entry log (5). Then the page log of the FPA page has to be deleted (6) to allow unlocking the FPA page (7). A transaction abort or system failure between operations (3) and (4) will cause recovery using the FPA page log information, as described in the previous section. Recovery after operation (5) will use the entry log to compensate the FPA modification by calling the FPA manager operations. It must no longer be affected by recovery actions of the corresponding transaction, because other transactions may change different entries of the FPA page, which would also be affected by the recovery.

(1)	locking of FPA page	
(2)	logging FPA page	(LOG-PAGE)
(3)	modification of FPA page	
(4)	locking of FPA entry	
(5)	logging of FPA entry	(LOG-MODIFY-FPA)
(6)	delete log information of FPA page	(DELETE-LOG-PAGE)
(7)	unlock FPA page	

Figure 23: MODIFY-FPA locking and logging operations

5. Implementation of the Transaction Support in LADY

In the previous chapter, we have discussed the transaction support for PRIMA and its implementation issues in a fairly detailed manner. Especially, we have separated local and global system components and their interaction. In order to illustrate the dynamic aspects of transaction processing in a server complex we sketch our run-time environment. Therefore, we briefly describe how transaction management is embedded into our system environment. PRIMA is implemented in LADY [WM85, WHB86], a language for the implementation of distributed systems, and uses the concepts and operations of the INCAS project [Ne87]. LADY distinguishes between three language layers:

- A *system* consists of a set of teams interconnected via ports. The interconnection structure is given by either directed port-to-port channels or by logical busses (multicast, broadcast).
- Each *team* (more precisely: each team type) is decomposed into several processes and monitors. Monitors are used for intra-team communication.
- At the *module* level the algorithmic behavior of each module type (process, monitor, class, procedure) is specified.

At the system level the complete NDBS prototype, i.e. PRIMA, the application layer, and the application programs, consists of a set of teams belonging to different team types. Asynchronous communication via ports is achieved by using sender and receiver processes together with a communication monitor.

The transaction management is implemented as one process with an interface monitor to provide an asynchronous interface. The TAMs at different nodes are connected by input and output ports with logical busses among them. On the other side the TAM provides its interface (asynchronous) to the data system, the access system and the storage system (fig. 24).

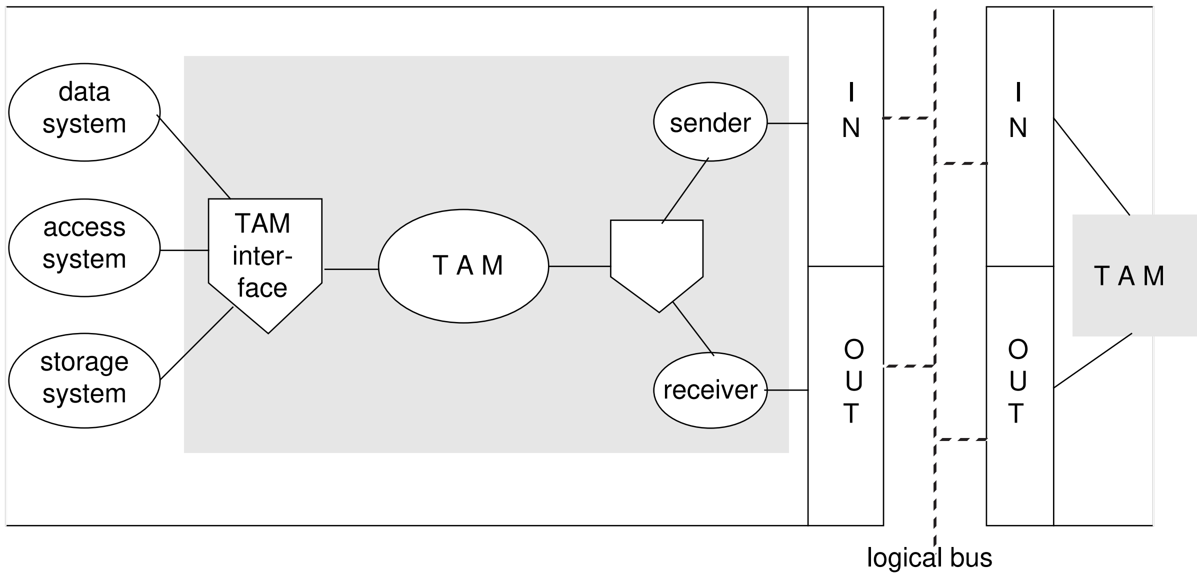


Figure 24: LADY structure of transaction management

Synchronization is done by one GLM and several LLMs, one at each PRIMA team. All LLMs communicate with the GLM via logical busses and ports. They are the asynchronous sender and receiver processes. The storage system (buffer manager, FPA manager) use the lock manager interface monitor (fig. 25).

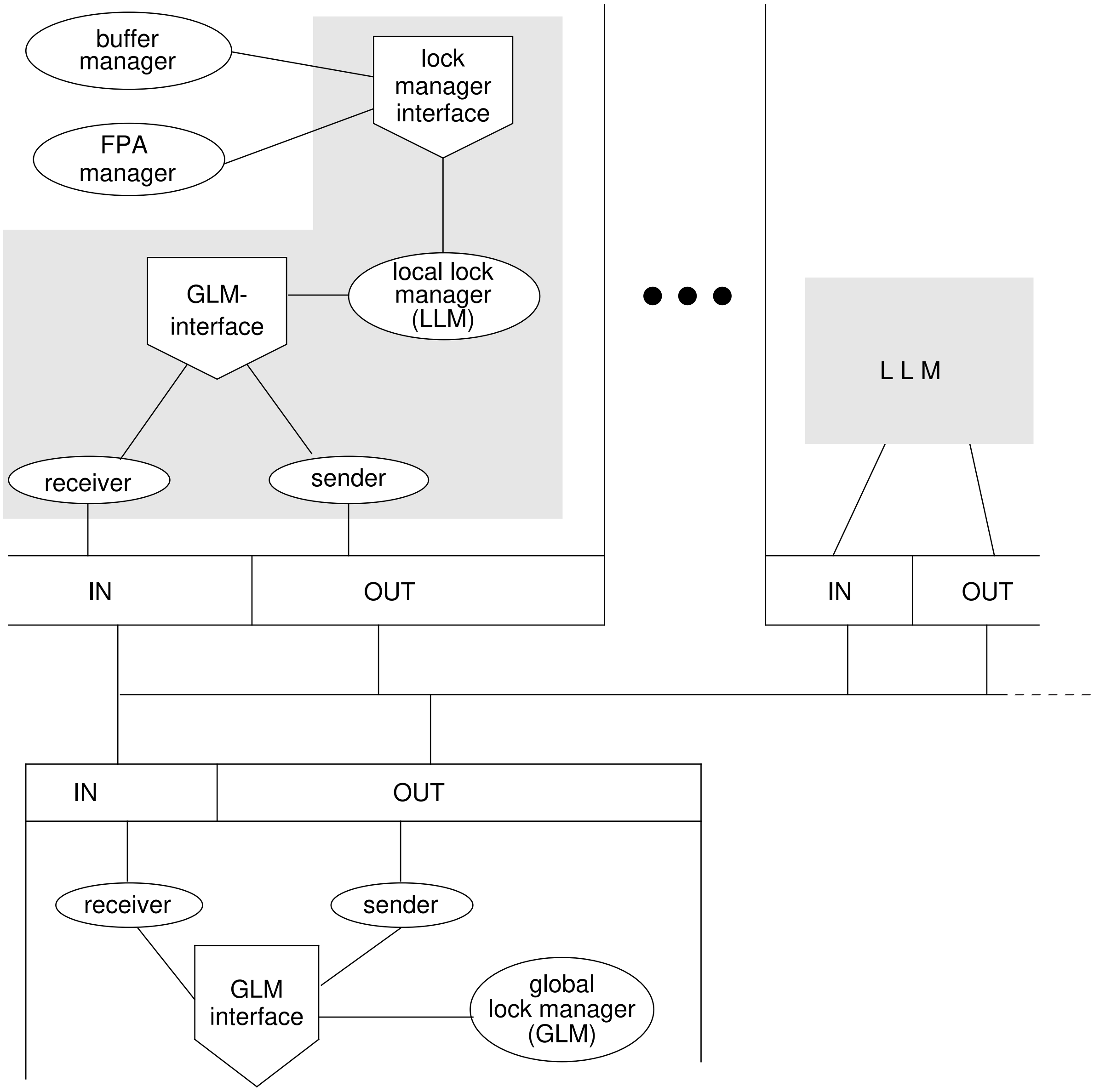


Figure 25: LADY structure of GLM and LLM

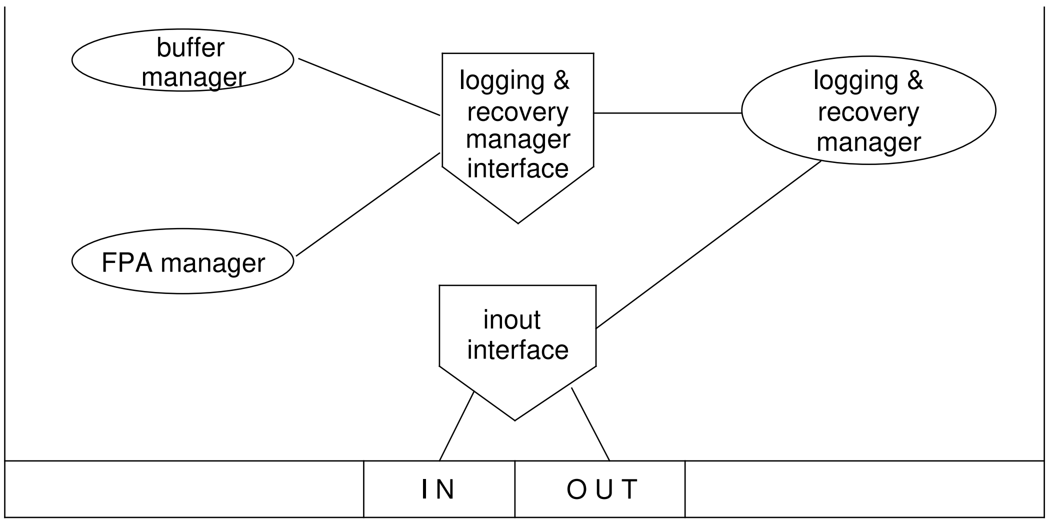


Figure 26: LADY structure of physical logging and recovery management

The logging and recovery managers at the PRIMA servers have no connection with each other. They use the inout interface monitor to the file system which is a separate team. The log and recovery functions are used by the buffer manager and the FPA manager via the logging & recovery manager interface monitor (fig. 26).

6. Summary and Outlook

This paper introduced the transaction concept of the NDBS kernel PRIMA, using open nested transactions in a distributed architecture, where nesting is also allowed within the abstraction levels. Open nesting for storage system transactions provides transparency of the buffer invalidation problem, caused by the loosely coupled architecture and of pages at the storage system interface. So the higher levels only see atoms as the basic objects for data processing, locking and logging. Distribution across processors is managed with an efficient mechanism for commit processing that saves messages and avoids nested messages.

The synchronization and recovery mechanisms for physical objects (pages and FPA entries) were introduced with special respect to low message flow for solving the buffer invalidation problem. On top of these concepts more abstract concepts may be provided for nested transaction implementation. Again transaction management in higher system layers is facilitated by dealing with higher-level objects; furthermore, it may be adjusted to the corresponding object properties thereby gaining higher system concurrency.

Modifications for the buffer manager strategies (NOFORCE/NOSTEAL instead of FORCE/ STEAL) are desired. A change from page logging to entry logging, which then had to be integrated into the access system, is interesting in order to reduce log information at the physical level.

Aspects for further investigation are the effects of closely coupled systems to logging and recovery, whereby lock management would also be affected. Further work will be extension of the transaction concept to the application layer and the investigation of higher lock and log granules, such as molecules or application objects.

References

As76 Astrahan, M.M., et al.: System R: A Relational Approach to Database Management, in: ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

As81 Astrahan, M.M., et al.: A History and Evaluation of SYSTEM R, in: Communications of the ACM, Vol. 24, No. 10, October 1981, pp. 632-646.

Ch87 Christmann, H.-P.: Ein Algorithmus zur Systempufferverwaltung und Synchronisation in einem lose gekoppelten Mehrrechner-Datenbanksystem, in: Proceedings of the GI/NTG Conference on Communication in Distributed Systems, Aachen, N. Gerner, O.Spamiol (ed.), Informatik-Fachberichte No. 130, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1987, pp. 412-425.

CS85 Christmann, H.-P., Schöning, H.: Simulation von Algorithmen zur Systempufferverwaltung und Synchronisation in einem lose gekoppelten Mehrrechner-Datenbanksystem, SFB124 Research Report No. 21/85, University Kaiserslautern, 1985

Fr87 Franz, B.: Konzeption und Implementierung eines Dateisystems für das DISTOS-Betriebssystem, Diplomarbeit, University Kaiserslautern, 1987.

Gr81 Gray, J.: The Transaction Concept: Virtues and Limitations, in: Proceedings of the 7th International Conference on VLDB, Cannes 1981, pp. 144-154.

Hä78 Härder, T.: Implementierung von Datenbanksystemen, Carl Hanser Verlag, München, 1978.

HHM86 Härder, T., Hübel, C., Mitschang, B.: Use of Inherent Parallelism in Database Operations, in: Proc. Conference on Algorithms and Hardware for Parallel Processing (CONPAR, Aachen, Sept. 86), Springer Verlag, Lecture Notes in Computer Science, Vol. 237, 1986, pp. 385-392.

HR83 Härder, T., Reuter, A.: Principles of Transaction Oriented Database Recovery, in: ACM Computing Surveys, Vol. 15, No. 2, 1983, pp. 287-317.

HR85 Härder, T., Reuter, A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen, in: Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Karlsruhe, März 1985, IFB 94, Springer Verlag, pp. 253-286.

HR87a Härder, T., Rothermel, K.: Concepts for Transaction Recovery in Nested Transactions, in: Proceedings ACM SIGMOD Annual Conference, 1987, pp. 239-248.

HR87b Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions, IBM Research Report RJ 5534, IBM Almaden Research Center, 1987.

HSS88 Härder, T., Schöning, H., Sikeler, A.: Parallelism in Processing Queries on Complex Objects, (Research Report, submitted for publication).

Ho88 Hoffmann, C.; Implementierung der Log- und Recovery-Komponente auf Seitenebene, Projektarbeit, University Kaiserslautern, 1988.

KLMP84 Kim, W., Lorie, R.A., McNabb, D., Plouffe, W.: A Transaction Mechanism for Engineering Design Databases, Proceedings of the 10th International Conference on VLDB, Singapore, 1984, pp. 355-362.

Kr88 Kreiselmaier, J.: Implementierung der Sperr-Komponente auf Seitenebene, Projektarbeit, University Kaiserslautern, 1988.

Li84 Lindsay, B., et al.: Computation and Communication in R*: A Distributed Database Manager, in: ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984.

Mi88 Mitschang, B.: The Molecule-Atom Data Model, in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB124 Research Report No. 26/88, University Kaiserslautern, 1988.

Mo81 Moss, J.E.B.: Nested Transactions: An Approach to Reliable Distributed Computing, PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, April 1981.

Mo82 Moss, J.E.B.: Nested Transactions and Reliable Distributed Computing, in: Proc. of the 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, Aug. 1982, pp. 33-39.

Ne87 Nehmer, J., et al.: The Multicomputer-Project INCAS - Objectives and Basic Concepts, in: IEEE Transactions on Software Engineering, Vol. SE-13, No. 8, 1987, pp. 913-923.

Pe86 Peinl, P.: Synchronisation in zentralisierten Datenbanksystemen, Algorithmen, Realisierungsmöglichkeiten und quantitative Bewertung, PhD Thesis, University Kaiserslautern, 1986.

Ra86 Rahm, E.: Buffer Invalidation Problem in DB-Sharing Systems, Research Report No. 154/86, University Kaiserslautern, 1986.

RS84 Reuter, A., Shoens, K.: Synchronization in a Data Sharing Environment, IBM San Jose Research Lab., preliminary version, (1984).

Sch88 Schöning, H.: The PRIMA Data System: Query Processing of Molecules, in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB124 Research Report No. 26/88, University Kaiserslautern, 1988.

Si88a Sikeler, A.: The Key Concepts of the PRIMA Access System, in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB124 Research Report No. 26/88, University Kaiserslautern, 1988.

Si88b Sikeler, A.: Buffer Management in a Non-Standard Database System, in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), SFB124 Research Report No. 26/88, University Kaiserslautern, 1988.

Ta76 Tang, C.K.: Cache System Design in the Tightly Coupled Microprocessor System, in: Proceedings of the AFIP National Computer Conference 1976, Vol. 45, pp. 749-753.

Tr83 Traiger, I.L.: Trends in Systems Aspects of Database Management, in: Proceedings of the 2nd International Conference on Databases (ICOD-2), Cambridge 1983.

WS84 Weikum, G., Schek, H.J.: Architectural Issues of Transaction Management in Multi-Layered Systems, in: Proceedings of the 10th Conference on VLDB, Singapore, 1984, pp. 454-465.

We86 Weikum, G.: Transaktionsverwaltung in Datenbanksystemen mit Schichtenarchitektur, PhD Thesis, Technical University Darmstadt, 1986.

WHB86 Wybranietz, D., Haban, D., Buhler, P.: Some Extensions of the LADY Language, SFB124 Research Report No. 26/86, University Kaiserslautern, 1986.

WM85 Wybranietz, D., Massar, R.: An Overview of LADY - A Language for the Implementation of Distributed Operating Systems, SFB124 Research Report No. 26/86, University Kaiserslautern, 1985.

YYF85 Yen, W.C., Yen, D.W.L., Fu, K. : Data Coherence Problem in a Multicache System, in: IEEE Transactions on Computers, Vol. C-34, No. 1, Jan. 1985, pp. 56-65.