Supporting Parallelism in Engineering Databases by Nested Transactions

T. Härder, M. Profit, H. Schöning
University of Kaiserslautern, West Germany

Abstract

Non-standard database systems supporting complex engineering applications are prime candidates for the exploitation of inherent parallelism in order to reduce the response time for user requests. Nested transactions are proposed as a control structure to achieve medium or even small grain parallelism. This paper investigates the design concepts and describes an implementation of the transaction management in a multi-layered database system running in a workstation-server environment. The tasks of the transaction management are structured according to the ACID principle: a transaction manager is responsible for the **A**tomicity property; furthermore it coordinates three components. A consistency manager achieves **C**onsistency preservation thereby allowing for deferred control or maintenance due to transaction nesting. A lock manager enables **I**solated execution where our locking protocol is designed for parent/child as well as sibling parallelism in a transaction hierarchy. Finally, a recovery manager takes care of various failures according to our extended failure model and accomplishes **D**urability of database updates.

1. Introduction

Non-standard database systems are designed to support interactive manipulation of complex engineering objects such as 3D-workpieces or VLSI chips. They are allocated on large server machines providing the data handling facilities for an entire design environment. On the other hand, engineering applications are already running on dedicated workstations well equipped with CPU power, memory and even private disks. Hence, effective and efficient workstation/server cooperation is needed for such applications.

Handling complex objects typically requires large numbers of object references to perform the construction work and to check the related integrity constraints. For this reason, special processing models exploiting as far as possible the principle of locality of reference are necessary to obtain satisfactory response times [Da88, HHMM88, SPSW90]. Existing design data must be able to be extracted at the server side on demand and transferred to the workstation where they are stored in an object buffer. This kind of special main memory structure provides fast operational access and a pointer-like reference mechanism for the application thereby guaranteeing "locality near by the application". Since design work may last quite a long time, copies of the design objects in the workstation may be preserved in a private DB to facilitate recovery or to save particular design states. Updates of the design data are accumulated in the object buffer until the design object is committed to the public DB. Such a processing model is controlled by a so-called design transaction [KLMP84], which may issue several checkout requests for data supply and one checkin request at the end of the transaction to commit the design work. Although a design transaction may perform additional local state changes in the workstation, only these two event-types (extraction and propagation of design objects) are important from a database server's point of view.

Checkout and checkin operations are directed towards sets of (different) complex objects. Given a high-level set-oriented database interface, such requests have to be specified by one or more DML operations. Each of these operations usually causes a very long execution path inside the DBMS since large granules of meshed structures representing complex design objects may have to be derived or maintained (bulk update during checkin). On the other hand, the interactive design application running in the workstation relies on short response times. Thus, minimal DBMS service times including fast checkout/checkin as well as effective transfer of design data should be guaranteed. In order to achieve a responsive request handling, the DBMS may decompose such sequences of powerful high-level operations into trees of suboperations and execute them using parallel processing concepts [Du87,HSS89, Lo88].

For the sake of a clean cooperation interface between workstation and server, each checkout or checkin operation has to be made atomic, that is, an appropriate transaction concept has to be provided. Moreover, concurrent execution of tasks on behalf of the same transaction requires sophisticated internal control structures allowing isolation when accessing shared data as well as maximal use of the inherent parallelism of the checkout/checkin operation. Hence, the chosen transaction concept should also support such processing structures in order to achieve medium or even small grain parallelism.

Objects of engineering databases are typically connected by multiple relationships of different types; all operations to derive or maintain complex objects have to use or to refer to these meshed structures very frequently, that is, smooth partitioning of data and operation is not possible (or extremely expensive). Therefore, distributed processing architectures based on loosely coupled processors (message communication) do not seem suitable, because frequent data access or data synchronization at a “remote” processor is very inefficient. So-called DB-sharing architectures [Sh85] also exhibit a number of severe disadvantages in the case of the parallel execution of a single DBMS request due to the possibility of database replication in the various DB buffers of the participating processors. Use of shared memory for critical functions such as data buffering, synchronization and logging is most important for performance reasons. Hence, we have designed our system to run on a server complex tightly connected by an instruction-addressable common memory used for critical, shared data structures. It provides memory-based message exchange as well as instruction-based synchronization primitives for shared data accesses (e.g. a “compare and swap” instruction).

In this paper, we are going to investigate the use of new transaction concepts and their implementation when exploiting parallelism in non-standard DBMS (NDBS). For this purpose, we introduce the essential concepts of an NDBS architecture and a model of NDBS operations [HMMS87, WS84]. In order to achieve a safe and efficient execution control for parallel actions, we sketch a model of nested transactions and tailor their properties to our NDBS environment. Subsequently, we discuss the functions of the transaction management, before proceeding to propose a multi-level synchronization mechanism and a component for integrity control adjusted to nested transactions. Furthermore, logging and recovery issues according to a refined failure model are considered together with appropriate implementation concepts. Finally, we conclude with a summary of our design proposals.

2. A Model of NDBS operations

In order to describe the kind of parallelism to be utilized and controlled, we introduce a model of our NDBS architecture. Such a model consists of multiple layers with well defined interfaces, as depicted in Fig. 1.

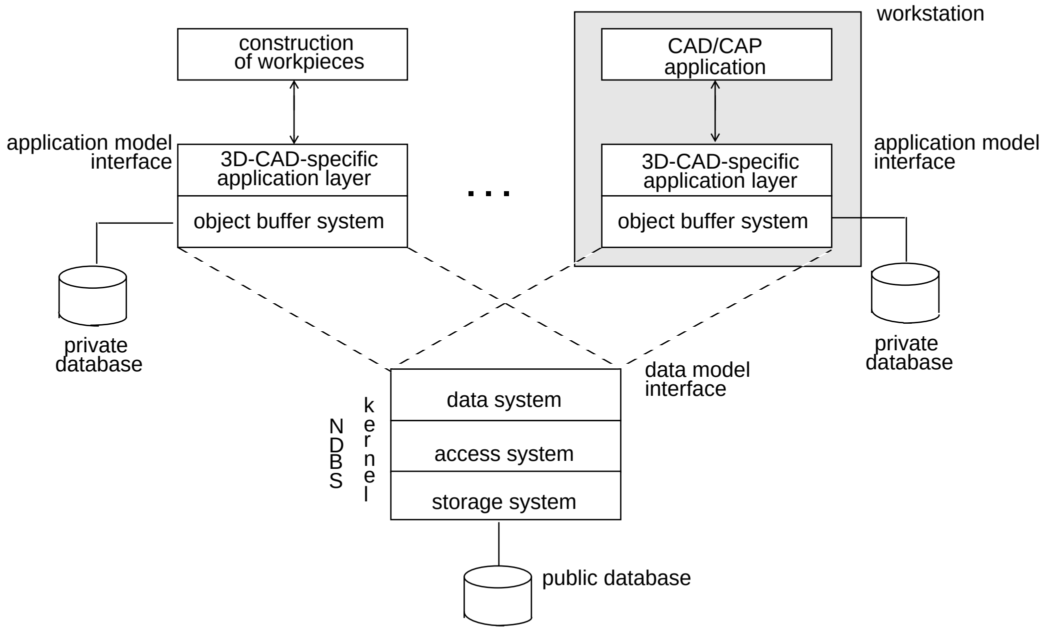


Fig. 1: Framework of an NDBS architecture

Applications and application-dependent layers of our NDBS architecture are allocated in a workstation environment; they are creating the workload we are dealing with, that is, checkout and checkin requests. This workload is processed by the NDBS kernel which is responsible for managing all aspects of the public database. Such a kernel is running on a server machine or a tightly coupled server complex to utilize parallelism more effectively.

2.1 Architecture of the NDBS kernel

For our purpose, it is necessary to refine our view of the architecture of the NDBS. It consists of a mapping hierarchy which embodies the major steps of dynamical abstraction from the level of physical storage up to the application model interface. At the bottom, the database is considered to be a very long bit string stored on disk which is interpreted by the NDBS. Proceeding from the bottom up, the objects become more complex with each level of abstraction. As a consequence, more powerful operations and more complex system-enforced integrity constraints are available on such objects at the interfaces. An important interface within the NDBS mapping hierarchy is the data model interface which is used to request kernel services. It supports an application-independent data model which allows the specification and manipulation of complex objects; in our case, we use the MAD model and the molecule query language MQL [Mi89]. On top of this 'neutral' NDBS kernel, two additional layers are provided to achieve application orientation of our architecture.

Our NDBS kernel is organized in three layers and only implements application-independent services and mechanisms [HMMS87]:

- The storage system provides a powerful interface between main memory and disk. It enables access to sets of pages organized in segments as well as managing a DB buffer. Furthermore, it is responsible for the appropriate mapping of database contents onto external storage and for the propagation of modified pages after their replacement in the DB buffer. Log information may also be buffered and written upon request.
- The access system manages storage structures for basic objects called atoms and their related access paths. The atoms of various types are connected to each other by bidirectional links of a specific type; these links represent the defined relationships of the database schema. The access system offers an 'atom at a time' interface to the next higher layer which uses operations for direct and sequential access along various access paths (scan operations). For performance reasons, multiple access paths and redundant storage structures may be defined for atoms.

- The data system builds complex objects (molecules) which are accessible at the data model interface. The molecules are specified by statements of the query language and are derived dynamically using the atom-oriented interface of the access system. For this purpose, the data system translates, optimizes and executes set-oriented DBMS requests. Molecule construction is based on the selection of atoms according to their link structure. Since molecules may be composed of sets of atoms of different types, the result set of a query usually contains structured objects with heterogeneous components. Such a result set is then delivered all together to the application layer (set-oriented delivery to reduce communication).

The NDBS components in the workstation use the services available at the data model interface to provide the uppermost interface of the NDBS. In contrast to the kernel interface, an application model interface is always tailored to a specific class of applications. Such application-dependent services may be implemented in the application layer by means of specific ADT's, that is, functions directly called by the application. These functions rely on the efficient use of database data 'near by the application'. Therefore, they refer to large main memory data structures managed by the object buffer system. Checkout operations are used to load the object buffer with design data extracted from the public DB whereas checkin operations propagate committed design data back to the public DB. These operations are specified by the object buffer system, typically using a sequence of MQL operations. Since there is a computer boundary between the object buffer system and the data system, the implementation of an NDBS as given in Fig. 1 has to efficiently cope with bridging these system layers. Hence, data transfer components on both sides have to guarantee a minimal number of communication requests and, as a consequence, set-oriented data supply. For this reason, all MQL statements belonging to checkout and checkin requests are delivered to the NDBS kernel at a time.

2.2 Aspects of Dynamic Processing

Thus far, we have outlined the static mapping of complex objects which is organized by the hierarchical layers of our NDBS kernel. In order to characterize the dynamics of query processing, we quickly sketch our execution model of the kernel. During the translation process in the data system, a query is transformed into an operator tree, which consists of nodes describing operators on complex objects and of edges, which indicate the calls/ data flow among operators. Such an operator tree corresponds to a kind of a procedural representation of a query in the data system. The leaf operators embody large numbers of access system calls to fetch single atoms or to scan along access paths. Such calls, in turn, may be characterized as trees of procedure calls in the access system. For some of these services, storage system requests may be necessary, e.g. to fetch some data pages from disk or to write some information to a safe place. Hence, these requests, in turn, create tree-like calling structures in the storage system.

A simplified view of the dynamics of kernel processing is illustrated in Fig. 2 ("Each call to a subroutine is an example of a primitive at one level of implementation invoking a set of primitives at a lower level of control [Da78]"). In conventional DBMS, such trees of suboperations are executed serially in a left-most depth-first manner, that is, each suboperation is synchronously invoked.

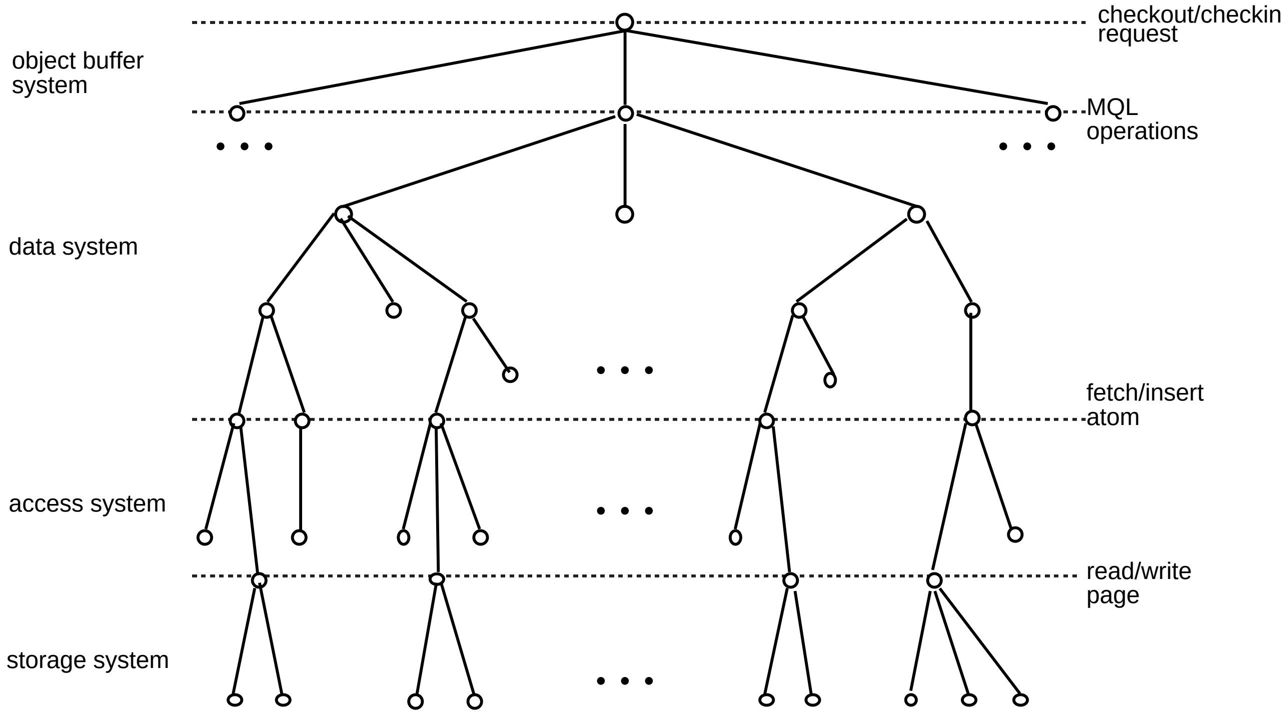


Fig. 2: Dynamic calling hierarchy in an NDBS kernel

A closer look at our workloads reveals that the operation trees spanned by typical requests may obtain a very large fan-out at each of our system layers, e.g. an atom type scan at the access system interface. Note, sets of molecules may have to be derived by a checkout operation and thousands of accumulated updates may have to be propagated to the database state by a single checkin operation. Accordingly, the construction or maintenance of each molecule require many access system calls. Hence, such broad operator trees promise effective optimization by using concurrent execution techniques. “Blind” creation of concurrent activities, however, is not desirable, because it would only increase data contention together with the related system overhead. Fortunately, we can use application semantics to decompose the database requests in such a way that concurrent requests do not conflict with each other at the level of decomposition. Furthermore, the optimizer in the data system may preplan subtrees of the operator tree which are independent of one another, to be executed in parallel [HHM86]. Such a non-partitioned approach to execute concurrent database requests is mandatory for strongly meshed data structures. For “delightful” data structures which can be easily partitioned, it should achieve the same results as obtained by a loosely coupled system.

Parallel execution requires an appropriate process and processor structure for our DBMS kernel. The system’s components, e.g. the various kernel layers, have to be mapped to a set of cooperating processes which may cooperate by asynchronous communication concepts. More specifically, cooperation among processes is achieved by a client-server mechanism; the calling process as the client (e.g. the data system) issues a request to the server process (e.g. the access system) which, in turn, may act as a client when calling a server (e.g. the storage system). To obtain maximum parallelism, our model allows for asynchronous processing, that is, a client may invoke one or more servers and proceed after the invocation (parent/sibling parallelism).

We assume that the choice of reasonable decomposition granules limits the useful degree of parallelism to typically less than 10^2. Furthermore, the amount of conceivable concurrent work varies from request to request. What is an appropriate scheme to map such kinds of parallel activities to our processor structure ($n \leq 30$ processors)? For performance reasons, we cannot rely on dynamic process creation. Therefore, our NDBS kernel is statically subdivided into n processes (or a few more). To respond to dynamic requests each of these processes performs multi-tasking. As opposed to expensive process switches this design decision permits fast switching of activities and cheap dynamic creation of new tasks within one process.

2.3 A Model of Nested Transactions

So far, we have identified various tasks of DBMS kernel processing which could be executed in parallel. The traditional control structure for DBMS processing (the transaction concept) is designed for serial execution of a user request. As the unit of atomicity, consistency, isolation, and durability (ACID principle [HR83]) it primarily guarantees concurrency transparency among transactions as well as atomicity of transaction execution despite the presence of failures (failure transparency). However, no specific execution control is performed within a transaction or within bulk requests such as checkout/checkin. The coordination of parallel and asynchronous activities within a single ’unit of work’ without explicit control structures would lead to complex programs that are susceptible to various kinds of failures. For these reasons, transaction processing requires some intra-transaction control structure to enable cooperation and isolation on shared resources and to limit the influence of failing activities.

The concept of nested transactions [Mo81] seems appropriate to achieve a safe and robust run-time environment for parallel and/or distributed processing within a transaction. Its main objective is the decomposition of a transaction into subtransactions which act as local control structures. Such a decomposition may be used to obtain suitable granules of concurrency control as well as access modes (locking modes) to enable maximum intra-transaction concurrency in a safe way. On the other hand, subtransactions may serve as units of in-transaction recovery which can be aborted and rolled back without any side-effects with regard to other transactions. Note with single level transactions, any failing activity will cause rollback of the entire transaction thereby possibly undoing large portions of transaction work.

Let us now quickly introduce our model of nested transactions, based on the framework and terminology developed by Moss [Mo81]. According to this model, a transaction may contain any number of subtransactions, and every subtransactions, in turn, may be composed of any number of subtransactions (see Fig. 3). The root transaction which is not enclosed in any transaction is called the top-level transaction (TL-transaction). Transactions having subtransactions are called parents, and their subtransactions are their children. Ancestors (descendants) of a transaction are reached by the reflexive transitive closure of the parent (child) relation. The set of descendants of a transaction C is called the sphere of C, as depicted in Fig. 3. If necessary, we will use the term superior (inferior) to refer to the non-reflexive version of ancestor (descendant). In the following, we will use the term ’transaction’ to denote both TL-transactions and subtransactions.

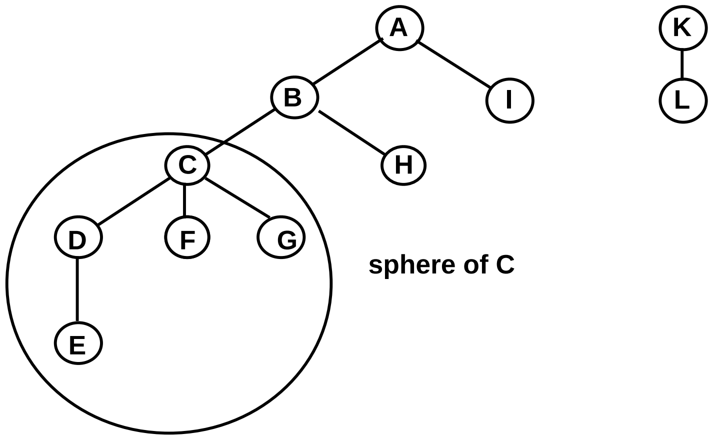


Fig. 3: Example of transaction trees

As already mentioned, single-level or flat transactions observe the ACID-principle. For nested transactions, these transaction properties should be realized for the TL-transaction (the outermost sphere of control [Da78]) which serves as an encapsulation mechanism against other TL-transactions, i.e. units of work belonging to other users.

As opposed to TL-transactions, it is sufficient to provide weaker properties for subtransactions. They can terminate either normally by committing or abnormally by aborting. However, a subtransaction works for its parent transaction which, in turn, works for the next higher transaction in the ancestor hierarchy. To enable flexible, but controlled cooperation within the transaction hierarchy, we require the following essential properties for nested transactions:

- Subtransactions can abort and must not affect the outcome of the surrounding transaction. Therefore, they must achieve an all-or-nothing type of execution, that is, atomicity is indispensable for subtransactions.
- Flat transactions preserve database consistency (DB-schema consistency). To require (this kind of) consistency to be preserved by every subtransaction would be unnecessarily restrictive and would prohibit many useful decompositions of a TL-transaction. In contrast, it is sufficient that a subtransaction running in a particular layer of our NDBS observes the layer-specific consistency constraints. Preservation of DB-schema consistency, however, may be delegated to transactions in the ancestor hierarchy, thereby allowing more flexible transaction decompositions and more effective consistency control.
- Subtransactions access shared data. Since they may fail (abort), their actions must be hidden from other subtransactions running concurrently. Hence, appropriate synchronization protocols (e.g. locking) are necessary to obtain isolated execution. On the other hand, cooperation requires that a transaction receives results (modified data) prepared by some of its inferiors. For this purpose, locks acquired by a transaction have to be passed on to the parent at transaction's commit. Such inherited locks may then be acquired by some transaction in the sphere of the parent transaction.
- Commit of a subtransaction and durability of its results are conditional subject to the fate of its superiors. Even if a subtransaction commits, aborting one of its superiors will undo its effects. All updates of a subtransaction become permanent only when the enclosing TL-transaction commits.

When associating Fig. 2 with Fig. 3, the model of nested transactions does not particularly restrict the use of subtransactions to enclose subtrees of our dynamic calling hierarchy. We may allocate multiple transaction levels within a single system layer or we may even bracket execution paths across layer boundaries. It is, however, a good design principle to observe layer boundaries within the transaction hierarchy. Therefore, we do not permit transactions to span multiple system layers. Hence, transaction properties (atomicity, isolation) may be used to control clean and safe cooperation across layer boundaries.

So far, we have introduced the properties of an abstract model of nested transactions. In order to investigate execution support for complex objects, we will tailor this framework to our specific NDBS architecture. A design transaction can be considered as a TL-transaction of our abstract model. It controls the engineering work at the workstation side and starts short transactions for specific tasks such as data supply and commit of results both of which are related to the public DB. These short (server) transactions belong to the NDBS kernel, they are, in turn, nested structures and are of prime interest to our objective. Therefore, we focus on transaction management at the server side.

Note, that these server transactions are exclusively responsible for all DB references/modifications of a design transaction which is the parent transaction of them. Issues of node autonomy and efficient transaction management dictate the allocation of an agent for the design transaction at the server side. An agent may be considered as a data structure keeping track of all checkout/checkin requests belonging to a design transaction. Such a concept isolates the server from workstation crashes, since it incorporates all information relevant for synchronization and recovery of public data. Furthermore, it facilitates rollback of transaction work when the effects of checkouts have to be undone at the server side. Each checkout/checkin is then modeled as a subtransaction of this agent. As explained in sect. 2.1, such a request consisting of multiple MQL statements is submitted to the server by the data transfer component where independent MQL statements could be invoked concurrently at the kernel interface. As indicated in Fig. 4, an MQL operation is guarded by an MQL transaction, which, in turn, may be decomposed into multiple data system transactions (depending on optimizer decisions to use parallelism). Typically, a data system transaction invokes many access system transactions each fetching or modifying a single atom. To organize and control parallelism, implies a certain amount of execution overhead which does not amortize when operation granules are too small. Therefore, we decided not to exploit fine granule parallelism possibly inherent in storage system calls and not to organize storage system calls as separate subtransactions (all transaction-related work like locking and logging is done at a higher layer). Hence, storage system operations may be considered to belong to the calling access system transactions.

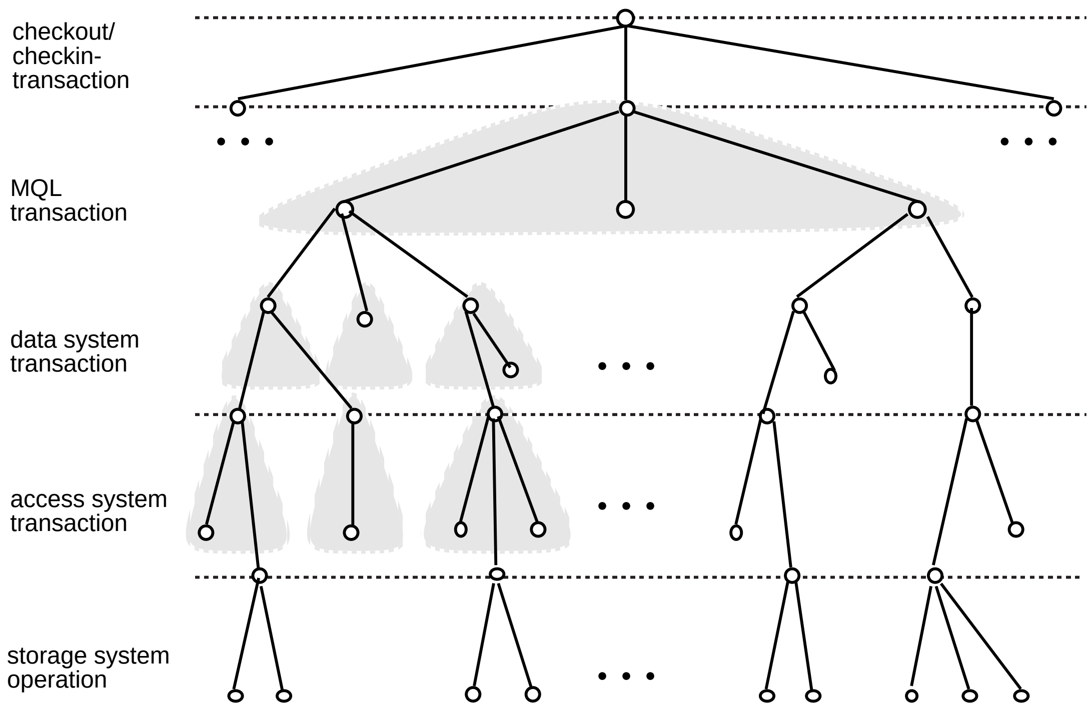


Fig. 4: Example of a transaction hierarchy in an NDBS kernel

In order to provide these nested control structures for the engineering applications, various services to maintain the principal transaction properties have to be accomplished. In our kernel implementation, we decided to group all transaction-related services in a so-called transaction management subsystem (see Fig. 5). It consists of four major components which are jointly responsible for preserving the ACID properties during transaction execution. Atomicity is achieved by the transaction manager (TAM) thereby relying on various other services. Consistency is controlled by the consistency manager (CM) whereas concurrency control performed by the lock manager (LM) takes care of isolated execution. Finally, durability of database updates is guaranteed by the recovery manager (RM) which collects log information to cope with various failure types. For performance reasons, the services of these managers are directly invoked by a transaction as long as it proceeds normally. In the case of special events, however, all managers have to be synchronized to do their job. Then, TAM is acting as a coordinator to distribute information or to obtain general agreement, e.g. during commit or abort.

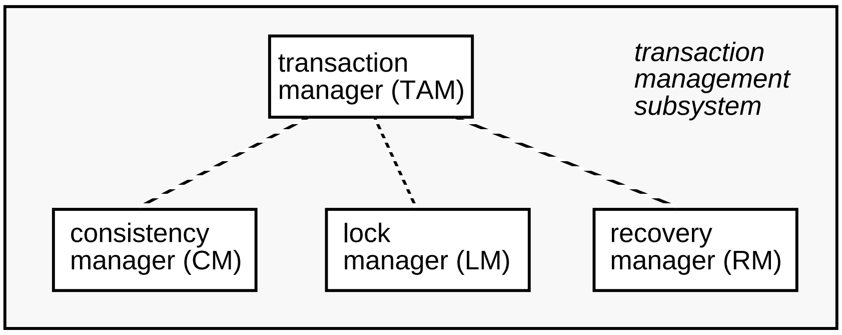


Fig. 5: Structure of the transaction management subsystem

3. Transaction Manager

Our prime objective is the investigation of parallelism in DB operations to be supported for engineering applications. For this reason, we have outlined a framework of nested transactions and have identified various transaction types according to our layered system model. In order to discuss the tasks of a transaction manager in more detail, and to propose mechanisms for its implementation, we introduce a number of functions for basic transaction manager services:

- Create_Transaction creates a subtransaction for the calling transaction.
- Commit_Transaction finishes a transaction thereby passing all relevant data related to transaction management to the parent transaction.
- Abort_Transaction rolls back the entire work of the corresponding transaction; all running or committed inferiors are also aborted whereas superiors are not affected by this operation.

3.1 Control Flow in a Transaction

Let us characterize the various tasks of transaction management by following the control flow of a transaction at the server side. As explained in sect. 2.3, a checkout request consisting of a set of MQL statements is passed on to the server. Before starting execution, the agent of the design transaction calls **Create_Transaction** to open a subtransaction, that is, to provide a sphere of control for checkout processing. Now, the checkout transaction may proceed; according to our decomposition scheme, it will issue a subtransaction for every MQL statement. Since such kernel requests may be independent, multiple MQL transactions may be created to run concurrently as separate tasks which may again create parallel subtransactions in the data system (subject to the MQL optimizer decision) and these, in turn, (parallel) access system subtransactions. Every data reference of such subtransactions has to be synchronized with appropriate data granules by using locking protocols (see sect. 4). Furthermore, data modifications have to be recorded by logging functions at the level of subtransactions to allow for fine granule failure recovery (see sect. 6).

When a transaction T has finished its computation (and all of its children are committed or aborted), it issues a **Commit_Transaction** to the TAM. According to our discussion in sect. 2.3, this operation plays a key role for the preservation of our transaction properties. Atomicity means that the TAM makes the results of T available to the parent transaction P after successful commit; otherwise the transaction T is aborted. Frequently, consistency constraints have to be controlled by the committing transaction T. If some of these consistency checks should be deferred (see sect. 5), commit processing will return appropriate reference information to the TAM component, which acts as a bookkeeper for the parent P. Isolated execution must not be limited to T, since its data modifications are preliminary (dirty) for transactions outside the sphere of the TL-transaction. The release of T's locks would sacrifice the serializability of the entire (TL-)transaction. Therefore, the parent transaction P inherits all locks from T and retains these locks for further usage. Durability of T's updates is also subject to the successful commit of all ancestors of T. Therefore, writing of data or REDO information to a safe place is not necessary at T's commit. To summarize, at **Commit_Transaction** of T, various types of information are transferred to the parent.

Even if the kernel layers are distributed across several processors, Two-Phase-Commit (2PC) [Gr78] for kernel subtransactions is not necessary, because durability is not guaranteed for them. In principle, this is also true for transactions spanning the workstation/server boundary, e.g. a checkout or checkin transaction. To enable isolated recovery from node crash at either side, however, we use a 2PC protocol for such a “distributed transaction” (see sect. 4.5). Commit of a checkin transaction, coincides with the end of the TL-transaction at the workstation site which enforces a 2PC protocol.

So far we have described the normal case. In the case of in-transaction failures (e.g. deadlock, consistency violation), a transaction might abnormally terminate. In contrast to a system failure (crash), where all main memory data are lost, **Abort_Transaction** is executed in an uncut DBMS run-time environment, all data belonging to the

DB state are available and operation-consistent [HR83]. Hence, the effects of the transaction can be wiped out using a kind of a reverse operation log. The transaction's resources are then discarded thereby restoring the transaction to its BOT state. As a consequence of an abort all active and committed inferiors are recursively aborted, too. After the completion of the abort processing the parent is informed of this abort.

3.2 Implementation of the Transaction Manager

It is obvious that this kind of processing will create highly dynamic transaction hierarchies embodying a lot of concurrent activities. In order to control these activities, the TAM has to perform some bookkeeping. First of all, the frequently changing transaction trees have to be maintained by suitable data structures. Moreover, it is necessary to collect all information related to synchronization, consistency control, and logging for the issuing subtransactions.

The nested transaction structure can be visualized by a set of m-ary trees where the nodes are transactions and the edges are parent/child relations. The root of such a m-ary tree corresponds to a TL-transaction. The transactions are represented by transaction control blocks (TCB) and the edges by pointers between them. All TAM operations described above lead to modifications of this data structure.

Our m-ary transaction trees are implemented by a special type of binary trees (see Fig. 6a). For efficiency reasons, we have added pointers linking children to their parent. Hence, a TCB contains three pointers (parent, child, sibling) which are used to establish the nested structures. In addition it keeps information describing identification, type (e.g. access system transaction), resources, etc. of the transaction. Fig. 6a illustrates the tree structures which correspond to our transaction trees of Fig. 3.

We now describe the effects of the TAM operations on the transaction trees. They are complemented by actions of the other components of the transaction management subsystem.

Create_Transaction(parent):child

allocates a new TCB and stores a system-generated transaction identifier, e.g. M (see Fig. 6b). Then, M is connected to its parent, e.g. B. Note, several transactions running in parallel may be created such that there is no relative ordering among siblings. Therefore, we always place the new TCB in the left-most position under a parent, as shown in Fig. 6b.

Prepare_Transaction(transaction)

performs phase 1 of a 2PC protocol (used for checkout/checkin transactions). To execute this operation the transaction must not have any uncommitted children. Depending on the outcome of the prepare phase, the transaction may be committed or aborted.

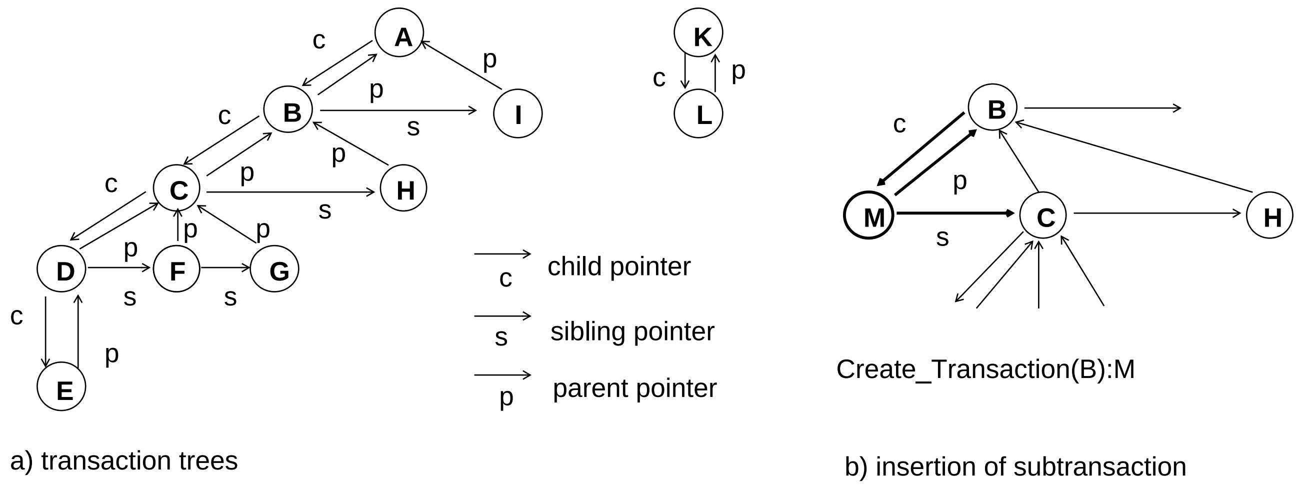


Fig. 6: Example of transaction trees

Commit_Transaction(transaction)

may only be performed if the transaction has no active children. Therefore in Fig. 6a, transactions E, F, G, H, I, L may commit, whereas D, C, B, A, and K may not. For example, if G commits, it is deleted from the child list of C and its TCB is destroyed. In the case of a checkout/checkin transaction **Prepare_Transaction** must have been called first.

Abort_Transaction(transaction)

walks through the list of the children belonging to the transaction, thereby using the child pointer and sibling pointers. For each TCB found in this chain an Abort_Transaction for the corresponding transaction is initialized, which again searches for children. This recursive process ends when no children are found anymore. For example, aborting transaction C leads to a depth-first abort sequence for the open transactions C, D, E, F, G. Abort of the transaction is completed as soon as the entire work is rolled back by the recovery manager (sect. 6.5) in reverse commit order of committed subtransactions.

Thus far, we have introduced our transaction manager, which is responsible for the manipulation of transaction hierarchies and for accomplishing the atomicity property of the transaction concept. As already mentioned, it acts as a coordinator of all transaction-related services. In this role, its main duties towards other components of the transaction management subsystem, include information supply to handle commit and abort events, as well as parent identification to perform functions related to inheritance or rollback.

4. Concurrency Control

An important aspect of transaction management is concurrency control which is performed by an extra system component. A large number of methods for concurrency control are known from the general DB literature, however, most of these algorithms do not seem to be effective in engineering applications. It is obvious that the typical reference behavior and the duration of such applications do not allow the use of optimistic protocols. Therefore, we have chosen a pessimistic method to be implemented in our NDBS kernel, that is, a two phase locking protocol (2PL) which had to be refined to the needs of concurrency control of nested transactions in multi-layered systems [We86].

4.1 Locking Rules

In the kernel, we follow a special hierarchical S-X lock protocol with share locks allowing for read access and exclusive locks allowing for write access and prohibiting all other (read or write) accesses, as described by locking rules for flat and nested transactions in [Mo81]. These read/write locking rules for nested transactions may be summarized as follows:

- A transaction may read (write) an object if it holds the corresponding lock in read (write) mode.
- A transaction may hold a lock in read (write) mode if all holders of the lock in write (any) mode are ancestors of the requesting transaction.
- When a transaction commits, its locks are inherited by the parent transaction which holds the lock in the maximum mode of already held locks and the inherited locks.

These locking rules are based on the assumption that only leaf transactions acquire and use locks (i.e. access the objects). Such a restriction, however, prohibits parent/child parallelism [HR87] and seriously limits the use of inherent parallelism. For this reason, we have extended the basic locking rules to enable maximum parallelism in a transaction hierarchy, that is, parent/child as well as sibling parallelism. We solved this problem by distinguishing between locks explicitly acquired (hold) and those acquired by inferiors and then passed on to their parent during commit (third rule). We call this kind of locks retained, because the transaction only retains them for further use by descendants. Retained locks do not include the right to access the object in the corresponding lock mode, but prohibit access of non-descendants to the locked object. For example, if transaction P retains an

X-lock, a transaction outside the sphere of P cannot hold a lock in any mode. Retained locks do not lead to conflicts between parent P and children, since P does not use them. If P wants to access an object, P has to acquire the lock by an explicit lock request. Hence, a transaction may hold and retain a lock on the same object. The held lock then was obtained by an explicit lock request, whereas the retained lock was inherited from a committed subtransaction. Once retaining a lock the transaction remains retainer of this lock until its end. As a consequence, our synchronization protocol avoids the structural restrictions of the original proposed by [Mo81].

By distinguishing the hold and retain states of locks the following modified locking rules are obtained:

- A transaction may read (write) an object if it **holds** the corresponding lock in read (write) mode.
- A transaction T may acquire a lock in read (write) mode if
  - no other transaction **holds** the lock in write (any) mode, and
  - all retainers of locks in write (any) mode are ancestors of T.
- When a subtransaction T commits, all (held and retained) locks are inherited by its parent P in retain state. If P already retains the lock, it keeps it in the more restrictive retain state.

As a consequence, when a checkout transaction commits, its locks are inherited by the agent of the design transaction in retain state. In order to increase failure isolation, these locks are kept in stable storage. (Hence, a workstation may not be bothered by a server crash, see sect. 6). To prevent failing of a checkin due to lock incompatibility, the design transaction has to acquire sufficient lock modes by its checkout requests, for all objects to be checked in. In the case, that the design transaction encounters the need for more restrictive locks for objects it has already checked out, it has to repeat the checkout with the appropriate lock modes (convert-only-request).

Synchronization of flat transactions requires serializability of them [EGLT76]. For nested transactions, this correctness criterion obviously holds among TL-transactions, too. But what correctness criterion should be applied for synchronization within the sphere of control of a transaction? Note, in order to avoid cascaded backout the uncommitted results of a transaction (and its sphere of control) must not be visible before transaction commit. This is guaranteed by our locking protocol. Hence, a parent or siblings of a transaction T may only see its updates after commit, that is they follow after T in an equivalent serial schedule. Recursive application of the following rules will produce a serializable schedule, that is equivalent to the result of the locking protocol:

1. A child is placed before its parent because it commits before the parent will commit.
2. Siblings may proceed concurrently keeping their locks until commit. Therefore commit of sibling_i before commit of sibling_j implies that sibling_i is placed before sibling_j.

According to these rules the following statements hold: Transaction A scheduled before another transaction B must never have seen modifications of B, whereas B might have seen the modifications performed by A. If A and B or their inferiors never accessed common data their relative order is not restricted and does not depend on the locking protocol. Assume, for example in Fig 6, that transaction I commits before B, H before C and D before F, then our serialization order corresponds to I, H, E, D, F, G, C, B, A. Now, our correctness criterion is serializability of transactions according to the above rules.

A locking protocol observing these rules enables sibling as well as parent/child parallelism. Further refinements of such protocols are discussed in [HR87].

4.2 Granularity of Locks

Until now, we did not consider the granularity of locks. In our system, the objects which will possibly be chosen for locking are those which occur in our layered architecture, that is, molecules, atoms, and pages. First of all, let us discuss why we ruled out molecules as locking granules:

- Each molecule has a type consisting of the atom and link types as specified in the molecule definition (e.g. the FROM clause of MQL) and an extension to be dynamically derived from the underlying atom network. Concurrency control at the type level would result in very coarse granules. Conflicts, e.g. incompatible overlaps of type structures, would be easy to detect, but such a policy would hardly be effective because of the frequency of fictitious conflicts.

- At first sight, direct locking of molecules seems to be an elegant and effective approach. However, it may be difficult because molecules do not have a static representation in the database. Furthermore, molecules are not restricted to hierarchies, but may have non-disjoint and graph-like structures. In order to discuss this general problem, we refer to the example in Fig. 7. The database schema corresponds to an atom type network with bidirectional link types. A molecule type definition selects a number of atom types and explicitly assigns an orientation to the participating link types. At the instance level, the database may be characterized as an (undirected) atom network where the molecule type definition is used as a template to derive the corresponding molecules (extensions). As illustrated in Fig. 7, even molecule types sharing some atom and link types need not embody common subhierarchies because the orientation of some link types may differ. When checking the two simple molecule types A-C-D and B-D-C, we detect an overlap on atom types C and D connected by link types C-D and D-C. Because of this overlap, locking of, for example, a2 and b2 as representatives of the molecules is not sufficient; further locks have to be acquired in the overlapping region. Even harder problems arise when comparing two molecules of the same type. It would be nice to lock the molecules of type A-C-D represented by the atoms a1 and a2 by locking the representatives (a1, a2). However, these two molecules overlap on c1 prohibiting this idea. Furthermore, locking of C type atoms (e.g. c1, c2) does not solve the problem, since there might be (and in this case are) overlaps with type D. As a consequence, molecules of the same molecule type may overlap unpredictably on each atom type. Even if two specific molecules do not overlap, some molecule of the same type could do so. For these reasons, it is hard to check locking conflicts at the molecule level. (Other approaches [He89] are based on an hierarchical data model which allows easy detection of common subhierarchies.) However, concurrency control for molecules may be performed by regarding all their building blocks. Locking a molecule means to lock its extension, that is, the set of atoms (and links) which make up the molecule. Hence, a set of atom locks will be used to represent a molecule lock.

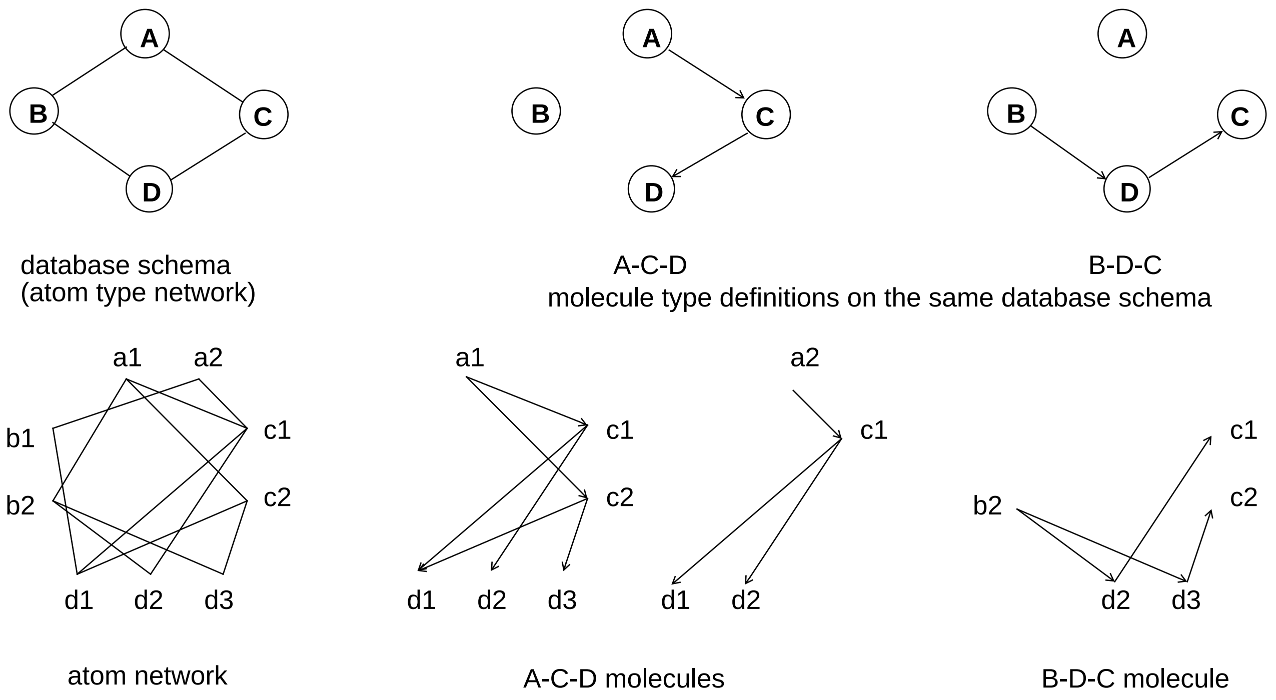
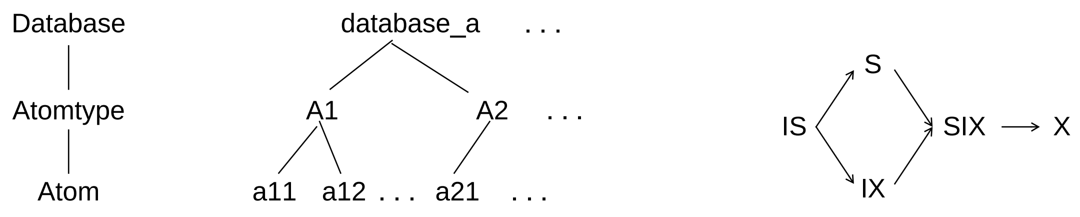


Fig. 7: Molecule types and extensions of the MAD model

Atom locks are similar to record locks in conventional DBMS. They are acquired in access system operations and are kept until the end of transaction. Hence, atoms represent our principal granule of locking. When accessing a page during an access system operation, a page lock must be acquired for each page to be accessed, in order to isolate the transaction from concurrent transactions accessing different atoms in the same page (implicitly acquired by the FIX operation). Such an isolation is necessary to accomplish physical consistency of the pages, if modifications or displacements of storage structures (atoms) are performed within the page. These page locks may be released at the end of a storage system call (UNFIX operation); the objects under consideration, however, remain protected by the atom locks. If page locks were held longer than the requesting access system transaction lasts and inherited along the transaction hierarchy this might cause serialization of subtransactions belonging to different subhierarchies within the TL-transaction. This would happen, if both subtransactions access atoms which reside in the same page. The second transaction would be unnecessarily delayed, thereby decreasing parallelism.

Atoms as lock granules are appropriate for transactions which only access a few objects, but cause high costs (space, number of lock requests, etc.) when a large portion of the database is accessed, e.g. by means of an atom type scan. Object hierarchies may reduce these costs [GLPT76, Gr78]. However, to use the object hierarchy efficiently, without reducing parallelism, new lock modes (so-called intention share/exclusive locks IS, IX) must be introduced [Gr78]. The key ideas are:

- Every object (node) in a hierarchy can be locked explicitly thereby locking its entire subhierarchy implicitly.
- Each node on the path from the root to the object has to be locked in the appropriate intention mode.



a) object hierarchy b) lock mode ordering

Fig. 8: Hierarchical object locking

A further refinement is the share and intention exclusive (SIX) mode which grants an S-lock for the entire subhierarchy; in addition, it indicates the intention that X-locks for finer granules may be explicitly requested later on. Locks are acquired along the path from root to leaf and are released in the reverse order (leaf to root). As a consequence of permitting partial ordering of lock modes (see Fig. 8b) a simple MAX computation is not sufficient in the case of multiple inheritance of locks on the same object. Therefore, an operation UNION on two modes M1, M2 is defined, computing the least restrictive mode including both modes (M1, M2). In the hierarchical S-X protocol the following computation serves that purpose:

$$\text{UNION(M1, M2)} = \begin{cases} \text{SIX} & \text{M1 = IX and M2 = S or} \\ & \text{M1 = S and M2 = IX} \\ \text{MAX(M1, M2)} & \text{otherwise} \end{cases}$$

4.3 Conflict resolution

Incompatible lock requests cause lock conflicts among transactions resulting in wait situations. For our locking protocol, we must distinguish between two cases: concurrency control among independent transactions (TL-transaction) and within a nested transaction hierarchy. An engineering transaction may last a long time and, therefore, waiting on resources of such a transaction does not seem appropriate. Hence, a lock request of a subtransaction in the sphere of a TL-transaction A which conflicts with a lock held by another TL-transaction K or its inferiors, (see Fig. 3) should not cause a lock wait, but should be resolved by notifying the requestor, explaining the reason for the rejection.

Serial and synchronous execution of the transaction tree does not provoke lock conflicts among subtransactions; in this case, locks only serve for isolation against interference from other TL-transactions. As soon as subtransactions accessing shared data are executed concurrently (either parent/child or siblings, in addition), synchronization is required inside the sphere of a TL-transaction. Hence, waiting situations may occur and, as a consequence, deadlocks among participating subtransactions.

4.4 Deadlocks

Because of the blocking nature of our lock protocol, deadlocks may arise (even among subtransactions). Since deadlock prevention seriously restricts parallelism, we have implemented a deadlock detection mechanism which resolves an existing deadlock by transaction backout. For this purpose, wait-for relations are maintained for transactions competing for resources. The resulting wait-for graph is represented by a matrix; whenever a lock conflict occurs, this graph is searched for cycles indicating deadlock situations. For simplicity, the transaction closing the cycle is chosen as a victim and rolled back (see sect. 6).

In nested transactions, another class of deadlocks may be observed, which cannot be detected by means of the standard wait-for graph. Let us explain the problem with a simple example. In Fig. 9a, part of a transaction tree is shown, where transaction B retains an S-lock on object O1 (denoted r:S(O1)). Now consider the case where I holds an S-lock on O2 and requests an X-lock on object O1, which is not granted due to the S-lock of B. Now, I waits for grant of the requested X-lock on O1 (acquire X(O1)). Furthermore, let H request an X-lock on O2 which forces H to wait for I.

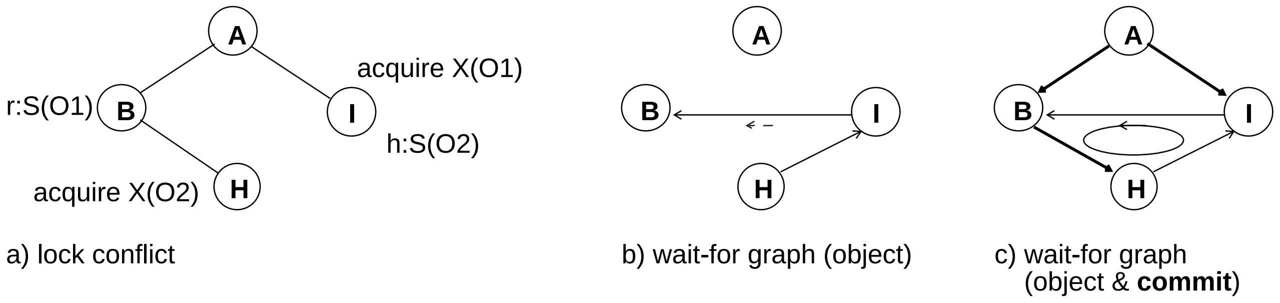

A
B
I
H
r:S(O1)
acquire X(O1)
h:S(O2)
acquire X(O2)
a) lock conflict
b) wait-for graph (object)
c) wait-for graph (object & **commit**)

Fig. 9: Deadlock inside a TL-transaction

At this point, we have a new form of deadlock, which is not detected in simple object wait-for graphs (Fig. 9b). The deadlock consists of three transactions with the following wait-for relations:

- H waits for I because of O2,
- I waits for B because of O1, and
- B waits for commit of H.

Obviously, such deadlocks are caused by different wait situations (wait-for-commit). In our example, the existing deadlock can be detected by adding a wait-for-commit edge from B to H to the wait-for graph. In general, each parent/child relation has to be added as an edge to the wait-for graph (Fig. 9c). Now, all deadlocks including those provoked by wait-for-commits will be detected. Obviously, it would not be a good decision to abort H in order to resolve the deadlock, because it could arise once again by restart of an equivalent subtransaction by B. The better choice is to abort B or I. For this reason, the deadlock resolution algorithm does not automatically chose the closing transaction when wait-for commit relations are involved. A transaction having no parent in the cycle (there is no wait-for-commit edge to it) will be chosen as the victim of deadlock resolution. However, avoiding a deadlock instead of deadlock detection and resolution is often the better mechanism since it saves work and therefore enhances efficiency. Under certain circumstances we are able to avoid deadlocks based on wait-for-commit relations and the conventional wait policies (FIFO). When we choose a more liberal granting sequence for the pending locks. Fig. 10 depicts the creation of such a deadlock situation where B acquires X(O) before H does so. This deadlock can also be detected by means of the introduced mechanism (Fig. 9c). However, if the lock requested by H is granted before that of B (a direct consequence of rule 1 which serializes all children before their parent), we will avoid such a deadlock. The example can be generalized to all inferiors of B. In general, a lock may only be granted if the transaction has no inferiors waiting for a lock on the same object.

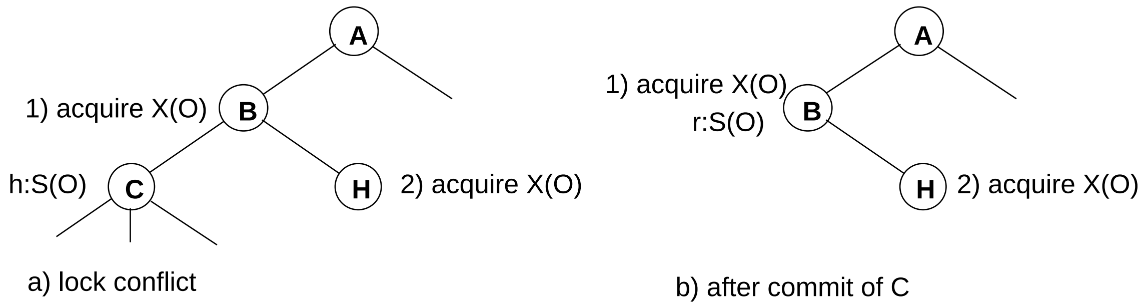


Fig. 10: Avoidable deadlock inside a TL-transaction

4.5 Lock Management

Concurrency control is performed by a system component called lock manager. It provides operations to lock and unlock objects, as well as to convert the mode of a granted lock. Furthermore, it enables lock inheritance at a child's commit, thereby assigning the lock in retain state to the parent.

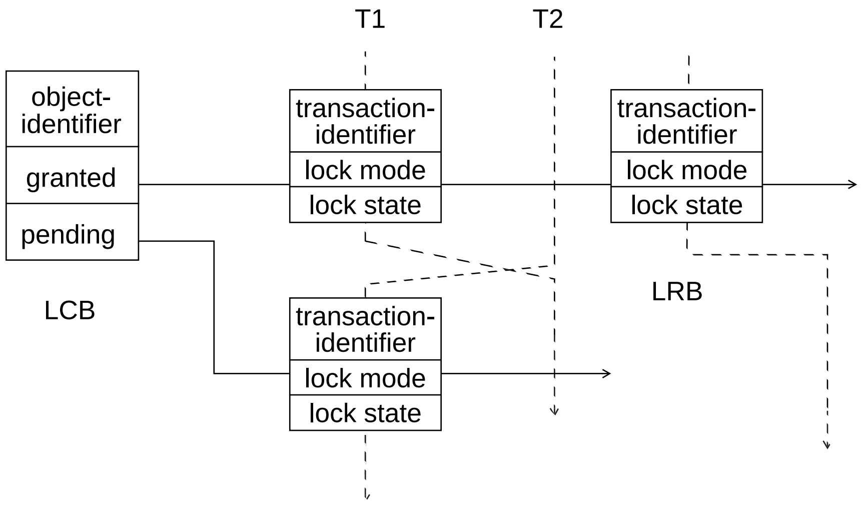


Fig. 11: LCB and LRB data structures

To manage the granted locks and the pending lock requests, the lock manager has to organize a kind of a book-keeping (see Fig. 11), which is for the most part based on information about object, requesting transaction, mode, and state of the lock. For this purpose, lock control blocks (LCB) representing the objects are allocated.

Now we are able to discuss the operations of the lock manager in detail:

Lock(transaction, object, lock_mode)

requests a particular access right on an object for the calling transaction. A hierarchical locking scheme is followed where appropriate intention locks are implicitly requested for higher-level objects. The implicit lock requests follow the same procedure as the explicit one. If the object is unknown to the lock manage a new LCB is created, whereas an LRB is always allocated. To check whether the request may or may not be granted t, we have to perform the following tasks:

- At first, we check whether the pending chain is empty; if not so we append the LRB to it in FIFO order to avoid starvation problems; otherwise we
- check for compatibility (according to the rules in sect 4.1) against all LRB's in the granted chain. If there are only compatible locks, the LRB is appended to the granted chain, otherwise it is appended to the pending chain.

Furthermore, the LRB is connected to the transaction's chain of locks. If the lock is granted the lock manager returns control to the calling transaction, otherwise the transaction is suspended and the wait-for graph is checked for deadlocks.

Unlock(transaction, object)

releases the lock held by a transaction on an object and all intention locks on the corresponding higher-level objects (bottom-up). It does so by searching the LRBs connecting the LCB of the object and the transaction and deleting them from both chains. (To guarantee a 2PL protocol, lock requests after the first **Unlock** operation of a transaction are not permitted). We proceed by checking and granting lock requests from the pending chain until the first lock request which is incompatible with one of the granted locks, is reached. Control is returned to the issuing transaction, after all possible lock requests (especially conversion requests) are granted (and the corresponding transactions are resumed)

Convert(transaction, object, old_lock_mode, new_lock_mode)

changes the lock mode of a transaction's object to a more restrictive one (new_lock_mode), usually from S to X. If the conversion can be granted the lock mode in the existing LRB is modified to new_lock_mode and the lock manager returns control to the calling transaction. Otherwise, an LRB for a lock request in new_lock_mode is created and inserted behind the last inferior's LRB of the converting transaction in the pending chain (children are serialized before parent) and the LRB is connected to the transaction's chain of LRB's. Because of the pending lock request the transaction will be suspended and a cycle test to detect deadlocks is performed on the wait-for graph. The request will remain pending until it is released from the pending chain by an **Unlock** or **Inherit** operation. Note, that a second conversion request of another transaction will immediately lead to a deadlock. As in the lock operation all intention locks of higher-level objects have to be converted first.

Inherit(child, parent)

passes all locks from a child transaction on to its parent. The lock manager does not need to know the transaction nesting, since the parent is told by a parameter. Usually this operation is invoked by the transaction manager during commit on behalf of the committing transaction. To maintain the lock data we have to

- change the lock states from hold to retain,
- add the chain of child LRB's to the parent, and
- to look for multiply retained locks preserving the strongest lock mode computed by the UNION function (see sect. 4.2).

Furthermore, we have to check all inherited objects for grantable lock requests. The procedure used is different from the one executed during **Unlock**. It does not strictly follow the order of the pending chain:

- First, we collect all inferiors of the parent transaction. This set of inferiors is sorted according to the transaction's position in the transaction hierarchy (children are serialized before parent). Then their lock requests are checked for compatibility with all locks granted so far. The process stops at the first LRB requesting an incompatible lock. Note, we only have checked inferiors for compatibility, whereas lock requests of transactions outside the sphere of the parent still remain pending.
- Furthermore, we start to check the pending chain from the beginning as described in the **Unlock** operation.

Clear_Locks(transaction)

releases all locks of a transaction by deleting the LRB chain of the transaction and by searching the pending chain of all affected objects to check if pending lock requests may be granted (the processing follows the description of the **Unlock** operation). This operation will be called when aborting a transaction or when committing a TL-transaction. Retain locks of superiors are not affected by this operation.

Stable_Locks(transaction)

writes all locks of a transaction to a save place (logging) where they can survive system crashes. Stable locks are necessary to isolate TL-transaction activities at the workstation side from server crashes. The operation will be invoked during the prepare phase of checkout transactions' 2PC.

5. Consistency Control

As mentioned in sect. 2.1, the layers of our NDBS kernel reflect different levels of abstraction. Each layer has its own interface, providing operations and guaranteeing integrity constraints which grow more complex with an increasing level of abstraction. For example, consider the access system interface. Here, operations on basic objects are supported, e. g. the deletion of an atom. To perform such a deletion, the data of the atom have to be removed from the corresponding page, addressing information has to modified, and the atom has to be deleted from all access paths concerned (as it is done in any commercial database system). One of the consistency constraints guaranteed by this operation is that all access paths have the same view of the database (*index completeness*). For example, if there is a B*-tree defined on attribute 1 and a B*-tree on attribute 2 of the same atom type, an atom of this type must be removed in both B*-trees.

The deletion of an atom forms a subtransaction consisting of various storage system operations, for example, to achieve the deletion of the atom from B*-trees. Thus, at the end of this subtransaction, the index completeness has to be guaranteed.

Analogously, consistency constraints guaranteed by data system operations can be identified: The deletion of a molecule is accomplished by a lot of atom deletions. The molecule structure is reflected in the links among these atoms. Links are represented by pairs of complementary "REFERENCE" attributes pointing to each other. All molecule operations guarantee that each REFERENCE attribute has its proper counterpart (*referential integrity)*. Again, a data system operation is a subtransaction consisting of many operations of the access system. In the case of referential integrity one could be in doubt as to which level of subtransaction should guarantee the constraint: Of course, one could enforce referential integrity after each atom deletion by correcting the proper counterpart. However, this would be an inadequate allocation of consistency warranty. Deleting two atoms which are referencing each other would lead to the modification of the second atom when the first one is deleted, followed by the deletion of the second one. Thus, unnecessary overhead would arise.

Conventional database systems integrate consistency constraints, such as the ones discussed above into the DBMS code. Some of them (e.g. [Es76]) allow for additional, user defined consistency constraints, which may or may not be bound to the transaction concept. We argue that integrity constraints of both kinds (inherent and user defined) should be controlled explicitly by a consistency manager, for the following reasons:

- Extensibility is one main requirement for NDBS in order to be able to keep up with the evolution of application areas. Not only new data types and operations, but also new access paths and new algorithms should be able to be integrated into the DBMS [CD86]. All these extensions may have an impact on the integrity constraints guaranteed by the various operations. Therefore, we suggest strongly a single place where all actions needed for consistency maintenance are centralized in order to make it more easily changeable.

- Scattering the inherent consistency checks all over the DBMS code does not reflect the boundaries imposed by a transaction concept. As in the case of immediate corrections of atom deletions discussed above, they might be integrated in an inadequate place. Furthermore, they are often not explicitly coded, but "ensured" to be never violated due to the structure of the DBMS code. This is a very dangerous approach in the case of extensible database systems, where access paths, data types and operations may be added to the system. It is very unlikely that later extensions take care of all integrity constraints "hidden" this way in the code.

- Proposals for user defined consistency constraints usually distinguish at most between three different times when a constraint has to be fulfilled: Immediately (i.e., after each operation), deferred (i.e. at the end of a transaction), or user driven. We exclude the user-driven check from our considerations, since it does not provide any consistency *guarantee* from the view of the system. The differentiation between deferred and immediate vanishes when a nested transaction model like ours is concerned: Since each operation invokes a subtransaction at a lower level, an immediate consistency constraint at one level is a deferred constraint from a lower level viewpoint. Thus, it is sufficient to handle only one kind of constraint, i.e. the constraints checked at the end of a transaction.

It should be clear now that every constraint can be related to a certain level in the transaction hierarchy, i.e. it holds at the end of every transaction at this level. We distinguish between two kinds of reaction when a constraint is not fulfilled at commit time:

- Some constraints allow for a system-driven correction. For example, referential integrity can be system-enforced. In this case, the commit is always successful.

- Other constraints cannot be enforced automatically. In this case, we do not abort a corresponding transaction, neither do we accept the commit. Rather, we reject the commit request thereby returning information as to why consistency is violated. Thus, the calling level may either perform corrections or may abort the subtransaction.

Now, the question arises as to how we can check consistency efficiently. Obviously, without further information, we would have to inspect the whole database, which is in general unacceptable. On the other hand, we only need to check those data which were affected by changes of descendants of the corresponding subtransaction. Thus, we must collect information about the changes of a subtransaction which is evaluated at the end of the transaction to check consistency. Sometimes, the subtransaction performing the changes runs at a lower level than the one which checks consistency. In this case, the collected information has to be passed on to the parent transaction at commit time.

These considerations lead us to a set of operations that have to be provided for consistency management:

When a transaction is started, the consistency constraints controlled by this transaction have to be specified, because they may vary depending on the layer the subtransaction belongs to. For this purpose, the command **Control_Constraints** is invoked by the **Bot** processing.

In the course of a transaction, data about consistency violations have to be collected by using the operation **Collect_Consistency_Information**. The corresponding information is gathered as a part of the subtransaction's resources.

During transaction **Commit**, before any other action is taken, the operation **Check_Consistency** has to be performed in order to check the state of the database against the consistency constraints guaranteed by the subtransaction. If this check fails, and consistency cannot be system-enforced, the **Commit** request has to be rejected. Otherwise, the data corresponding to the checked consistency constraints are released, while other data (belonging to constraints guaranteed by a higher-level transaction) are passed on to the parent transaction.

As a consequence of the **Abort** of a transaction, there must be an **Abort_Constraint_Control** command to release all consistency violation data collected in the course of the transaction which is now aborted. This includes all data that have been passed to this transaction by their children.

We assign the management of consistency control to a specific component of the transaction management, called consistency manager. More details about its design and its extensibility can be found in [Schö90].

6. Recovery Manager

Up to this point, we have described our approach to ensure the transaction properties of consistency and isolation. These properties represent the “good” case of a transaction, where no failure occurs. The durability aspect as well as the atomicity aspect, however, which are important whenever an error occurs, still have to be considered [KHED90].

6.1 Failure Model

Before discussing recovery issues, we must outline our failure model, that is, the failures to be anticipated by our NDBS. For this reason, we introduce several classes of failures to be dealt with:

1. A **transaction failure** is said to have occurred when a subtransaction is aborted for any reason, e.g. deadlock. In this case, the effects of the transaction and of its inferiors have to be wiped out (as already described in the previous sections), whereas the transaction’s superiors are not directly affected. Recovery can use the current system state to undo the work of the transaction tree under consideration, since no data are lost (in contrast to the failures we will discuss in the following). In the framework of our NDBS, this is not a user-initiated event, and hence should not appear too frequently, if the DBMS code is carefully designed. The recovery action invoked is called transaction UNDO.
2. A **communication failure** occurs if a message is lost or garbled and the underlying communication system cannot correct this failure. We do not consider communication failures appearing among the processors of our server, since we assume all message exchange within the kernel to be shared-memory based and therefore not error-prone. Nevertheless, we have to handle failures in the workstation-server communication, i.e. during the 2PC among these components.
3. A **crash** may either affect

   a) some but not all processors of the server, or

   b) the whole server, or

   c) the workstation, or

   d) server and workstation.

   In the cases b), c), d) the contents of the main memory of the server or workstation, respectively, will be lost because it is volatile. Therefore, recovery has to restore the database using only information from secondary storage.
4. A **media failure** causes loss of data stored on a particular secondary storage device. They have to be recovered using data stored on other devices (archive recovery). The corresponding recovery action is termed global REDO.

The transaction properties dictate that the effects of all incomplete transactions must be removed from the database, whereas the contributions of all successful transactions must be properly reflected in it. Hence, failures which do not affect the NDBS run-time environment may be handled by a local NDBS action, e.g. subtransaction rollback followed by normal continuation of transaction processing. Other failure types demand a kind of system restart that will accomplish the most recent transaction-consistent database state after successful recovery [HR83].

6.2 Mapping Concepts for Updates

To understand our recovery concepts, we sketch our design decisions of key issues related to the way updates are mapped to the materialized DB (on disks). The run-time environment of our kernel (Fig. 1) consists of a large DB buffer (several MBytes) and a small log buffer (several KBytes), both residing in (volatile) main memory. For the propagation of modified pages to the materialized DB, we use an update-in-place policy called ¬ATOMIC [HR83] (the indirect mapping of ATOMIC policies seems to be too expensive for our applications). Since these schemes do not allow indivisible propagation of a set of pages, they are vulnerable to system crashes leaving the materialized database in an unpredictable state after a crash. Hence, physical logging is required (as opposed to logical or operation logging when an operation-consistent database state can be ensured).

Our large DB buffer enables us to keep all modified pages in it (or provides for an appropriate overflow facility to disk) at least until the end of the (checkin) transaction (¬STEAL property) facilitating transaction rollback. Since dirty pages (with uncommitted updates) are never propagated to the materialized DB, it is not necessary to adhere to a write ahead log protocol (WAL [Gr81]). On the other hand, we do not flush the modified pages to disks neither at end of subtransaction nor at end of checkin transaction (¬FORCE). Since crashes are considered (extremely) infrequent events, we adopted an optimistic policy with delayed propagation in order to achieve fast response times for the user. EOT for checkin just requires the system to force-write sufficient REDO log information from the log buffer to a safe place (temporary log file). The modified pages of a transaction are then replaced by normal buffer management operations in combination with a checkpointing scheme. In the course of these actions, REDO information is collected in the archive log enabling media recovery.

To summarize our recovery concepts, we have chosen a ¬ATOMIC, ¬STEAL, ¬FORCE scheme for the implementation of the recovery manager. It is based on direct mapping of updates which avoids the I/O overhead of indirect mapping schemes (e.g. shadow pages). No disk logging is required for UNDO purposes, and REDO only entails entry logging at transaction commit. Hence, our scheme reduces as far as possible synchronous log and data I/O in the transaction processing path thereby minimizing response time. Other mapping- or recovery-related actions such as checkpoints or page propagation are removed from the transaction path and are asynchronously executed.

6.3 Log Information

Our database recovery concept, like all provisions for fault tolerance, is based on the use of redundancy. Since database operations are exposed to different failures, different kinds of redundancy should be available for cost-effective and efficient recovery [Gr81, RM89].

Operation log

In-memory rollback is performed by keeping a trace of inverse operations at the atom level in a special log structure (operation log) in a log buffer; obviously, it may be discarded after checkin transaction commit.

Temporary log

Crash recovery, in our case, requires recording of all REDO information to be kept in a temporary log file on disk; this log has to be used for partial REDO whenever committed results are lost due to a system crash. Hence, this information can be released as soon as all modified pages of a transaction have been stored on disk, e.g. after the next (direct) checkpoint. An important observation concerning the log granule determined our design decision: The lock granule must be at least as large as the log granule, otherwise unilateral rollback of one transaction may cause some interference with the normal work of others [Hä87]. In order to enhance parallelism (see sect. 4.2), we decided to use atom locking. As a consequence, we had to choose physical entry logging which records the storage structures of atoms or their fields as log information (e.g. for REDO operations). Other reasons supporting this decision were economic storage use and greatly reduced I/O overhead due to log page buffering.

Archive log

To perform global REDO after a media failure, we need an archive copy of the database and an archive log which must contain all changes committed to the database after the state reflected in the archive copy. Concerns with respect to fault tolerance sometimes dictate the keeping of multiple DB copies and the use of duplex logging. A media failure usually means the repetition of days or weeks of work by using the archive log. Since entry logging may be too slow for this purpose, we decided to rely on page logging for media recovery. In order to avoid disturbance of normal database processing, we designed a special algorithm to accomplish transaction-consistent archive logs based on checkpoint processing.

6.4 Recovery actions

Now we are ready to discuss the various recovery actions necessary to cope with the expected events according to our failure model. Since some of these failures may have an impact workstation-server cooperation, we should make some remarks beforehand on the recovery measures at the workstation side; an in-depth discussion may be found in [HHMM88].

The overall objective of workstation-server cooperation is node autonomy and strict isolation of processing according to the client-server principle. Mutual masking should be achieved as far as possible for all failures; a failure at the server side should not bother the workstation and vice versa.

As already mentioned, cooperation is exclusively performed via checkout/checkin transactions which are protected by 2PC protocols. Hence, mutual dependencies may occur while such requests are being executed. We carefully designed private recovery measures for the workstation side in order to minimize failure propagation [HHMM88]. Since they were tailored to a single-user environment, we could accomplish a simple, but efficient implementation. The key concepts are

- recovery points that are established by the system at the beginning and end of recovery transactions; restart after a workstation crash reconstructs the workstation state at the last complete recovery point.
- savepoints which serve as a means for user-initiated rollback to reach a previously marked design state; hence, a user may apply a restore operation to wipe out his work back to such a user-controlled savepoint.

We have achieved mutual masking of failures to the largest extent possible, by encapsulating each checkout/checkin request by a recovery transaction. Hence, recovery points enclose cooperation requests to the server.

Now, we are able to begin the discussion of our recovery actions at the server side.

Transaction failures

¬STEAL implies that transaction rollback is always a (fast) main memory operation. By using the operation log, transaction UNDO proceeds until all effects of the transaction and its inferiors are removed from the current DB state.

The design transaction in the workstation may restore its state as kind of a partial transaction backout. Reestablishing such a previously marked savepoint may cause the “rollback” of checkout requests, that is, the corresponding locks kept by the design transaction’s agent have to be released.

Communication failures

Most errors are assumed to be dealt with by the underlying communication system. The communication protocol of our NDBS may use inquiry messages if expected events are late and allows for the retransmission of messages between workstation and server. Communication failures in the server due to unrecoverable memory locations are handled like system crashes.

Crashes

In the case of a processor failure (case a), we have not implemented dedicated recovery measures. In our tightly coupled environment, a processor failure typically results in a (running) system state having invalidated data structures as well as pending locks, etc. which would complicate recovery to be performed by a surviving processor. Therefore, a processor failure is treated like a server failure.

A server crash (case b) requires partial REDO of all committed checkin transactions for which propagation of all modified pages cannot be assured, that is, for checkins finished since the last checkpoint (¬FORCE). By analyzing the temporary log the last checkpoint and all transactions which may have lost some committed data are determined. After that, all database pages not reflecting the most recent transaction-consistent state are recovered by the corresponding temporary log data. Restart processing is then completed by the creation of a checkpoint. A crash interrupting a checkout or checkin transaction leads to a complete loss of the transaction work (¬STEAL). Therefore, all information about such a transaction is lost after successful restart. However, the cooperation protocol between workstation and server guarantees continuation of work, e.g. by retransmitting the request after an inquiry message. Completed checkout of running design transactions represented by agents in the server are not affected by a server crash, since their data structures (locks) are kept on stable storage.

A workstation crash (case c) interferes with server processing only if it occurs in the middle of a checkout/checkin request. These are rare events in a design transaction. Private workstation recovery will reestablish the latest recovery point which coincides either with the beginning of a local recovery transaction or of a “critical section” under consideration. In the former case, the server is not affected, whereas some special server actions may be necessary in the latter case. Depending on the restart time, the server may have decided to abort the corresponding transaction because communication was interrupted for a long time. Hence, retransmission of the request will invoke repeated execution. If the workstation is up again for 2PC processing, a workstation crash may not even be observable at the server side. On the other hand, if a crash interrupts the 2PC protocol, the server may reinitiate 2PC processing after successful restart of the workstation or enforce retransmission of the request together with repeated execution. Hence, only limited disturbances are visible at the server side.

Finally, we have to handle the hopefully extremely infrequent situation where server and workstation crash together (case d). If it occurs outside a critical section, then independent recovery measures are applied in both environments. A double crash interrupting checkout/checkin processing implies crash recovery of the server (¬STEAL prevents UNDO work) as well as of the workstation. After successful restart, the workstation will reinvoke the corresponding transaction.

Media failure

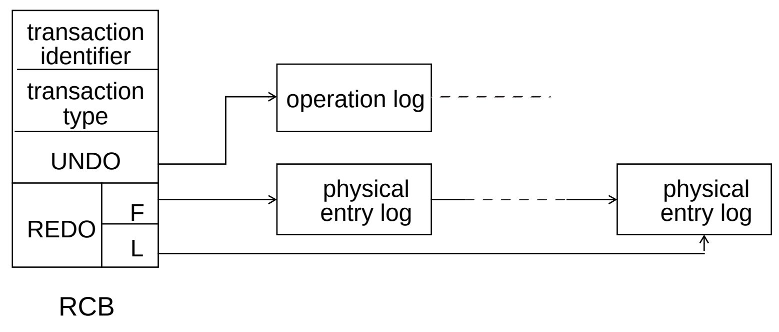
Loss of “permanent” data requires archive recovery which may last some hours or more. Global REDO of lost data has to be performed by using archive copy and archive log. Here, we don’t want to go into the details of this complicated procedure.

So far, we have described the recovery actions implemented to “cure” the various failures to be anticipated according to our failure model. Since restore operations to savepoints are supported at the workstation side, a kind of a recovery action may be necessary at the server side. If checkouts are “undone” by establishing an earlier savepoint, server recovery has to assure that the agents of these checkouts disappear; as a consequence, the stable locks belonging to these events have to be released.

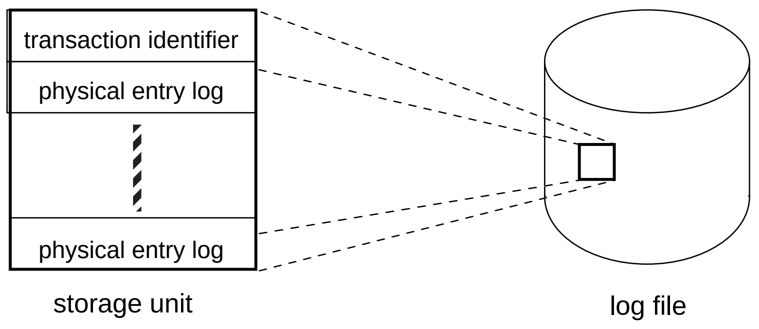
6.5 Recovery Management

All issues related to logging and recovery are performed by a system component called recovery manager. It keeps suitable data structures for log information tailored to the requirements of nested transactions [RM89] and provides recovery algorithms handling the various failure types. The recovery manager interface to transaction management may be outlined by the following operations:

Before describing the operations we will introduce the data structures used by them. One can distinguish between data structures only used in main memory and those designed for secondary storage (Fig. 12). For each transaction, a recovery control block (RCB) is allocated which stores the transaction identifier, transaction type and anchors for operation log (UNDO) and temporary log (REDO). To write the REDO-log efficiently to secondary storage, we allocate storage units which may buffer several physical log entries; they are asynchronously written to secondary storage whenever they are filled. At commit, they have to be propagated in a synchronous way.



a) main memory data



b) secondary storage

Fig. 12: Data structures used by the recovery manager

Bot(transaction, type)

announces a new subtransaction to the recovery manager. A recovery control block (RCB) is created and a storage unit for disk writes is additionally allocated if it is a checkin transaction.

Log_UNDO_Operation(transaction, operation, parameters)

inserts a log entry for an UNDO operation at the beginning of the UNDO-log. During **Rollback** the chain is processed starting at the beginning.

Log_Atom_Afim(transaction, operation, identifier, attribute_list)

installs a log entry for insert, delete, and modify operations on the specified atom. For insert and modify operations an attribute list is required, which contains the affected attributes. Logging of delete operations only needs the atom identifier. The new log entry is added to the end of the transaction's REDO-log in order to preserve the order of operations.

Inherit_Log(child, parent)

moves all logging information of the child to its parent transaction (compare to Inherit_Lock sect. 4.5). To preserve the sequence of update operations we insert the UNDO-log at the beginning of the parent's UNDO-log and add the REDO-log at the end of the parent's REDO-log. If the parent is a checkin transaction, the log entries are copied to its storage unit. Whenever the checkin transaction's storage unit is filled it will be asynchronously written to disk, thus enhancing efficiency of the **Prepare** operation.

Prepare(transaction)

starts the 2PC protocol thus forcing all log data not yet written (storage unit) together with a prepare record to disk. The transaction must be a checkout/checkin transaction. In the case of a checkout transaction no data log has to be written, but writing log information to preserve stability of locks (**Save_Locks**) is necessary.

Commit(transaction)

writes a commit record for the checkout/checkin transaction to disk in order to complete the 2PC protocol. At the end of the operation the REDO- and UNDO-log can be erased in main memory.

Rollback(transaction)

aborts a transaction within the sphere of a transaction. To do so, we process the UNDO-log of the transaction and delete the RCB of the transaction. In case of a checkin transaction we have to delete from disk the log data already written. Rollback of a committed checkout transaction implies the release of its stable locks managed by the design transaction’s agent.

Save_Locks(transaction, lock_list)

is invoked by the lock manager during its Stable_Locks operation, which belongs to the 2PC protocol of the transaction manager. The locks are immediately written to disk.

Crash_Restart()

restores the last checkpoint, reads the temporary log from disk and repeats the modifications made by committed checkin transactions since this checkpoint. The locks requested by committed checkout transactions are installed again. At the end the temporary log is deleted and a new checkpoint is created.

7. Conclusions

We have presented an investigation of the concept and implementation of nested transactions. The focus of our paper has primarily been on supporting parallelism in a multi-layered NDBS kernel by the use of nested transactions. Furthermore, the transaction mechanism had to be tailored to the cooperation requirements of a workstation-server environment.

Our transaction model was adjusted to the particular features of NDBS which consist of a multi-layered kernel and an application layer. The decomposition of subtransactions was largely guided by the kind of operations in engineering databases and these, in turn, were influenced by the layered architecture which determined the deepness of nesting. As a result, we obtained very broad transaction trees of moderate as well as fixed maximum depth. These trees controlling granules of work in a user operation lent themselves to concurrent execution in an efficient way. The invoking of transactions asynchronously enabled parent/child as well as sibling parallelism.

Although any distributed hardware architecture may be chosen to gain true parallelism, we rely on a tightly coupled multi-processor architecture for our NDBS kernel implementation. The reference behavior of engineering applications when operating on complex objects exhibits large degrees of locality on meshed data structures which does not permit efficient partitioning of data. Hence, shared memory is a prerequisite for the manipulation of such objects. Furthermore, shared memory greatly facilitates the design and implementation of nested transaction management.

We have grouped all transaction-related services in a transaction management subsystem which consists of four major components responsible for the essential transaction properties (ACID paradigm). The transaction manager administrates the transaction states by maintaining a forest of transaction trees; it is acting as a coordinator for the remaining components and performs commit or abort processing, i.e. achieves transaction atomicity.

We have chosen a hierarchical locking scheme for practical reasons (multi-granularity locking) and have tailored it to the needs of concurrent execution controlled by nested transactions. To obtain fine locking granules, we decided to apply atom locks to isolate concurrent operations on molecules. A novel aspect of concurrency control arises from the parallelism within a transaction's sphere of control for which a conflict resolution strategy had to be provided. Deadlock detection was improved by an enhanced wait-for graph where the regular wait-for relationships were complemented by wait-for-commit relationships. Furthermore, granting of locks deviated from the “fair” FIFO policy in special cases thereby reducing the probability of deadlocks (deadlock avoidance).

The notion of consistency incorporated in conventional DBMS had to be adjusted to the concept of nested transactions. It turned out that deferred consistency constraints in one layer of our kernel architecture are “immediate” ones in a higher layer. Hence, our consistency manager could take advantage of the nested transaction structure to control these constraints.

Our recovery mechanisms embody a kind of an optimistic attitude, that is, crashes are anticipated very infrequently and therefore synchronous logging or recovery-related actions in the transaction execution path are avoided as far as possible. For this reason, we have chosen a ¬ATOMIC, ¬STEAL, ¬FORCE scheme for the recovery manager which works with an in-memory operation log for UNDO and relies on disk-based entry logging for REDO. Hence, even commit operations for large complex objects require just a few synchronous log page writes as the only disk outputs.

Another objective of failure handling was the mutual masking of failures in a workstation-server environment. Perfect isolation was achieved by carefully designing the actions of a workstation design transaction and by using controlled cooperation with the server. The main concepts used were the encapsulation of checkouts/checkins by recovery transactions at the workstation side and a 2PC protocol to transfer requests/results. Furthermore, agents of checkout/checkin transactions were kept in the server which also possess the corresponding locks on stable storage.

In this paper, we have explored the concepts of nested transaction management for an NDBS kernel in sufficient detail. Problems of transaction management in a workstation environment such as recovery transactions, savepoints, cooperation in a group, etc. will be discussed in a subsequent paper.

Acknowledgements

The comments of C. Iochpe, E. Rahm and G. Weikum are gratefully acknowledged.

8. References

CD86 Carey, M.J., DeWitt, D.J.: Extensible Database Systems, in: On Knowledge Base Management Systems, Brodie, M.L., Mylopoulos, J. (eds.), pp. 315-330.

Da78 Davies, C.T.: Data Processing Spheres of Control, in: IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

Da88 Dadam, P. et al.: Managing Complex Object in R^2D^2, HECTOR-Project, Volume II: Basic Projects, Springer-Verlag, 1988, pp. 304-331.

Du87 Duppel, N. et al.: Progress Report #2 of PROSPECT, Research Report, University Stuttgart, 1987.

EGLT76 Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System, Commun. of ACM, Vol. 19, No. 11, 1987, pp. 624-633.

Es76 Eswaran, K.P.: Aspects of a Trigger Subsystem in an Integrated Database System, in: 2nd Int. Conf. on Software Engineering, 1976, pp. 243-250.

GLPT76 Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.: Granularity of Locks and Degrees of Consistency in a Shared Data Base, in: Proc. IFIP Working Conf. on Modelling in Data Base Management Systems, North-Holland, 1976, pp. 365-394.

Gr78 Gray, J.: Notes on Database Operation Systems, in: Operation Systems: an Advanced Course, Springer-Verag, LNCS 60, 1978, S. 393-481.

Gr81 Gray, J.N. et. al.: The Recovery Manager of the System R Database Manager, ACM Computing Surveys, Vol. 13, No. 2, 1981, S. 223-242.

Hä87 Härder, T.: On Selected Performance Issues of Database Systems, in: Proc. 4th GI/ITG-Conference on Measurement, Modelling, and Evaluation of Computer Systems, Erlangen, Sept. 1987, Springer-Verlag, IFB 154, pp. 294-312.

He89 Herrmann, U., et al.: A Lock Technique for Disjoint and Non-Disjoint Complex Objects, Fern-Universität Hagen, Informatik Berichte, Nr. 85, 03.1989, 15 p.

HHM86 Härder, T., Hübel, Ch., Mitschang, B.: Use of Inherent Parallelism in Database Operations, Februar 1986, in: Proc. of CONPAR86 Conference, Aachen, Sept. 1986, Lecture Notes in Computer Science, Vol. 237, Springer-Verlag, 1986, S. 385-392.

HHMM88 Härder, T., Hübel, Ch., Meyer-Wegener, K., Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server, in: Data & Knowledge Engineering 3, 1988, pp. 87-107.

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. VLDB 87, pp. 433-442.

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

HR87 Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions, IBM Research Report RJ 5803, San Jose, Calif., Aug. 1987.

HSS89 Härder, T., Schöning, H., Sikeler, A.: Parallel Query Evaluation: A New Approach to Complex Object Processing, in: IEEE Data Engineering, Vol. 12, No. 1, March 1989, pp. 23-29.

KHED90 Küspert, K., Herrmann, U., Erbe, R., Dadam, P.: The Recovery Manager of the Advanced Information Management Prototype, in: Reliability Engineering and System Safety 28 (1990), pp. 187-203.

KLMP84 Kim, W., Lorie, R., McNabb, D., Plouffe, W.: Nested Transactions for Engineering Design Databases, in: Proc. 10th Int. Conf. on VLDB, SIngapore, Aug. 1984, 355-362.

Lo88 Lorie, R. et al.: Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience, IBM Research Report RJ 6165, San Jose, Calif., 1988.

Mi89 Mitschang, B.: Extending the Relational Algebrra to Capture Complex Objects, in: Proc. 15th VLDB Conf., Amsterdam, 1989, pp. 297-306.

Mo81 Moss, J.E.B.: Nested Transactions: An Approach to Reliable Computing, M.I.T. Report MIT-LCS-TR-260, M.I.T., Laboratory of Computer Science, 1981.

RM89 Rothermel, K., Mohan, C.: ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions, in: Proc. 15th VLDB Conf., Amsterdam, 1989, pp. 337-346.

Schö90 Schöning, H.: Preserving Consistency in Nested Transactions, in: Proc. HICSS-23, Volume II, Hawaii, Jan. 1990, pp. 472-480.

Sh85 Shoens, K.: The AMOEBA Project, Proc. IEEE Spring Comput. Conf., San Francisco, 1985, pp. 102-105.

SPSW90 Schek, H.-J., Paul, H.-B., Scholl, M.H., Weikum, G.: The DASDBS Project: Objectives, Experiences, and Future Prospects, IEEE Transaction on Knowledge and Data Engineering, Vol. 2, No. 1, 1990.

We86 Weikum, G.: A Theoretical Foundation of Multi-Level Concurrency Control, in: Proc. ACM SIGACT-SIGMOD: Symposium on Principles of Database Systems, Cambridge, March 1986, pp. 31-42.

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