Institut für Informatik der Universität Zürich

Transaction Management in CoOMS

Andreas Geppert
Stefan Scherrer
Klaus R. Dittrich

Nr. 93.10 April 93

Transaction Management in CoOMS

Andreas Geppert
Stefan Scherrer
Klaus R. Dittrich

Technical Report 93.10

Forschungsbereich Datenbanktechnologie
Institut für Informatik
Universität Zürich
Winterthurerstr. 190
CH-8057 Zürich
Switzerland
Email: {geppert, scherrer, dittrich}@ifi.unizh.ch

May 4, 1992

“The transaction concept is an unrealizable ideal.” (J. Gray, [Gray 81])

Abstract

This paper summarizes the features and properties of transaction management in CoOMS.

Contents

1 Introduction

CoOMS [Dumm 91] is the database management system (DBMS) realizing the NO^2 data model [Dittrich 90, Geppert 90a, Geppert 90b]. As usually, if (multiple) users or applications would perform work with a DBMS (say, CoOMS) in an uncontrolled manner, inconsistent database states or incorrect views of applications or users might be the consequence. While the system cannot prevent users from violating consistency in all cases, inconsistencies due to incorrect interleaved executions or failures of any kind have to be avoided.

Usually, database systems provide for the concept of *transaction* in order to avoid inconsistencies of the last two kinds (failures and incorrect multi-user access) (see [Gray 78, Haerder 83, Bernstein 87, Reuter 87, Weikum 88]). Usually, a transaction is regarded as a *partial order* or *sequence* of database operations (and other operations, which however are not of interest from the viewpoint of the DBMS). Overall consistency is achieved in that the DBS guarantees the *ACID* property of transactions [Haerder 83]:

- **A:** *Atomicity* (failure atomicity). Transactions are atomic when they behave according to the "all-or-nothing" principle, i.e. all of the operations of any transaction are executed, or none of them.
- **C:** *Consistency.* Transactions are units of consistency: while inconsistent database states may occur *during* the execution of a transaction, consistency has to hold at the end of each transaction again.
- **I:** *Isolation*. This property says that transactions cannot "see" intermediate states from other transactions, thus are isolated from each other.
- **D:** *Durability.* Once a transaction has terminated, all its effects (i.e. modifications to the database) have to be permanent. Even if some crash or failure occurs later on, changes of a succesfully terminated transaction must not get lost.

Clearly, users or applications have to tell a DBS which operations belong to a specific transaction, i.e. when the transaction starts and when it finishes. For that matter, specific commands *begin_transaction*, *end_transaction* (or *commit)* and *abort_transaction* (or simply *abort*) are defined. *Commit* specifies the succesful termination of a transaction while *abort* denotes the unsuccessful end. In the first case, modifications to the database states have to become permanent, while in the second one no changes at all may become durable. Thus, a transaction is subject to *rollback* in the latter case. Rollback means that all changes initiated by the transaction in question have to be undone, i.e. the transaction does not affect the database state.

In the classical model (as initially described by Gray [Gray 78]), transactions may only execute simple database operations like *select, update*, and so on, but not transactions again. When focussing on that restriction, they are also called *flat transactions*. This restriction implies some weaknesses:

- *modularization* of units of work is not adequately possible,
- usually there is no possiblity for transaction internal checkpoints, i.e. no mechanism to roll back only some of the transaction's operations.
- transaction internal parallelism is not possible in a controlled manner.

Nested transactions ([Moss 81, Moss 85, Moss 86]) overcome these weaknesses in that they allow transactions to execute *subtransactions.* Subtransactions, in turn, may spawn subtransactions themselves, or execute simple database operations. With nested transactions, units of work can be structured or modularized; for that matter, each “subtask” (recursively) is modelled by a subtransaction. Checkpointing transaction-internal database states is easily possible, since subtransactions can be undone (rolled back) by means of *abort,* i.e. the database state at *begin_subtransaction* is a checkpoint. Furthermore, transaction-internal parallelism is possible in principle[1]. Actually, the kind of nested transactions as proposed in [Moss 81] only increases parallelism among subtransactions (and thus leads to better *intra-transaction parallelism* [Weikum 88]), while parallelism between top-level transactions is not affected (thus, *inter-transaction* parallelism is not improved when compared to the classical model).

This paper is organized as follows: in the next section, we define relevant concepts, namely (strictly) serializable schedules of flat transactions, recovery principles, and (*closed*) nested transactions in more detail. This section also covers another kind of nested transactions: open nested or multi-level transactions, which are used for internal synchronization and are useful in layered architectures. In section 3, techniques for the realization of the concepts proposed in section 2 are described. Section 4 summarizes the features of transaction management proposed for CoOMS. Finally, the last section elaborates on special (architectural) issues in CoOMS. Throughout the next two sections (concepts and techniques) we will discuss principles and alternatives in a general way, but keep the setting of CoOMS in mind, as far as intended application domain (including transaction profiles to be expected), transaction structures, and the corresponding data model are concerned.

2 Concepts

2.1 Consistency-Preserving Transactions and Schedules

As stated in the introduction, the purpose of transaction management is to ensure consistency when multiple transactions execute concurrently and various kinds of failures can occur. In this section, we give a formal notion of interleaved, however correct, executions of multiple transactions. We first restrict ourselves to flat transactions and elaborate on nested transactions later (in fact, issues and problems are not so different for nested transactions). A widely accepted criterion for correct interleaved executions of transactions at least in “standard” application domains is *serializability* ([Gray 78, Bernstein 87]).

Usually, the (DML) operations any transaction may issue are classified in *read* and *write* operations. Additionally, there are specific operations *commit* and *abort*. Without further knowledge (e.g. on concrete types of database entities and their semantics), a *write* operation on an object conflicts with any other operation on the same object (*read* or *write),* while *read* does not conflict with other *read* operations (it is said to *commute* with itself). A classification beyond *read/write* (e.g. for operations of abstract data types [Schwarz 84]) is possible and does not affect the notion of serializability essentially; the crucial point here is the notion of *conflict* [Korth 83]. In the following, we use the following abbreviations: we write w for a write operation, r for a read operations and op for any operation, respectively. C is a shorthand for commit, while a abbreviates abort. Finally, if we like to stress that an operation op is executed on behalf of transaction i, we subscript op with i (op_i).

1. In fact, nested transactions were initially proposed for reliable computing in distributed, thus parallel, systems.

Definition 2. 1 (Transaction)
A transaction T is a partial order $(\Sigma_T, <_T)$, where
1. Σ_T is a set of operations and $<_T$ a partial ordering $\subseteq \Sigma_T \times \Sigma_T$,
2. either $commit \in \Sigma_T$ or $abort \in \Sigma_T$,
3. for all operations $op \in \Sigma_T \setminus \{commit, abort\}$:
   $op <_T commit$ or $op <_T abort$,
4. $<_T$ orders at least the conflicting operations in Σ_T.

"Real life" transactions usually are (strictly ordered) sequences of database (and other) operations. However, from the viewpoint of transaction management, we are only interested in database operations. Furthermore, serializability theory still holds for the weaker definition of transaction as a partial ordering, and therefore would allow us to exploit transaction-internal parallelism. Additionally, note that we assume each transaction to come to an end eventually; i.e. we exclude failures for the moment.

In order to define serializability (or any other correctness criterion), we need a notion of execution sequence of operations from multiple transactions. Those interleaved transactions are represented by *schedules*.

Definition 2. 2 (Schedule)
A *schedule* is a triple $(T, \Sigma, <)$, where
1. $T = \{T_1,..., T_n\}$ is a set of transactions,
2. $\Sigma = \cup_i \Sigma_{Ti}$ is a set of operations,
3. $<$ is a partial order on Σ,
4. $\cup_i <_i \subseteq <$.

Following the definition above, we assume that each transaction terminates, either by means of *commit* or *abort.* However, that need not be the case when a failure occurs and some transactions have been executed only in part so far. Therefore, the notion of *partial schedule* (or *prefix*) is required.

Definition 2. 3 (Partial Schedule)
A *partial schedule* S' to a given schedule S is a triple $(T, \Sigma', <')$, where
1. $\Sigma' \subseteq \Sigma$,
2. if $op_1, op_2 \in \Sigma'$ and $op_1 < op_2$, then $op_1 <' op_2$,
3. for each $op_1 \in \Sigma, op_2 \in \Sigma'$: if $op_1 < op_2$, then $op_1 \in \Sigma'$.

A (partial or complete) schedule is serializable (i.e. correct in our sense), if it is equivalent to a serial one. Thus, two further definitions are required:

Definition 2. 4 (Serial Schedule)
A schedule $(T, \Sigma, <)$ is *serial*, iff $\forall\, T_i, T_j, i \neq j$:
either $\forall\, op_i \in \Sigma_i, op_j \in \Sigma_j: op_i < op_j$
or $\forall\, op_i \in \Sigma_i, op_j \in \Sigma_j: op_j < op_i$

Definition 2. 5 (Equivalence of Schedules)
Two schedules $S = (T, \Sigma, <)$ and $S' = (T', \Sigma', <')$ are *equivalent*, iff
1. $T = T'$,
2. $\Sigma = \Sigma'$,
3. $\forall\, op, op' \in \Sigma$, op and op' conflict: $op < op'$ iff $op <' op'$.

Now, serializability is easy to define: any schedule that is equivalent to a serial one is correct. Since both execute the same transactions (with the same sets of operations), and both order conflicting operations in the same way, they result in the same database state whenever they have started in the same database state. Consequently, interleaving transactions in a serializable fashion does no harm to consistency. The only difference between a serial and an interleaved execution is the ordering of non-conflicting operations; however, since those commute, they can be executed in any arbitrary order.

Definition 2. 6 (Serializable Schedule)
A schedule S is serializable iff it is equivalent to a serial one.

Serializability prevents inconsistent database states due to incorrect interleaving executions. Unfortunately, incorrect states may still occur in the presence of failure or transaction aborts. Therefore, if one of the transactions aborts or fails, other transactions may be forced to abort as well (*cascading abort*). This is the case whenever a transaction writes an object, another reads it, and then the former aborts *after* the second has read the object. Another problem occurs when a transaction writes an object, another transactions writes as well, and then the former aborts. In this case, atomicity requires to reinstall the object state that has been actual before the first transaction wrote. Thus, the abort of the first transaction will overwrite the second. Consequently, for practical systems (i.e. in the presence of aborts) we have to strengthen serializability. The stronger criterion is called *strict serializability* and simply requires that no object may be read or written until the transaction that wrote it most recently either commits or aborts.

Definition 2. 7 (Strict Serializability)
A serializable schedule S is strictly serializable, iff
$\forall\, w_i \in \Sigma_\mathrm{i}, op_j \in \Sigma_\mathrm{j}, \mathrm{i} \neq \mathrm{j}: w_i < op_j \Rightarrow c_i < op_j \vee a_i < op_j.$

Another, but equivalent way to characterize serializability is by means of *serialization graphs*. In such a graph, nodes represent transactions while edges express the fact that operations between the connected transactions conflict.

Definition 2. 8 (Serialization Graph)
A serialization graph for a schedule (T, Σ, <) is a directed graph (T, E), such that

1. $\mathrm{E} \subseteq \mathrm{T} \times \mathrm{T}$,
2. $e = (\mathrm{T_i}, \mathrm{T_j}) \in \mathrm{E}$ iff
   $\exists\, op_\mathrm{i} \in \mathrm{T_i}, op_\mathrm{j} \in \mathrm{T_j}: op_\mathrm{i} < op_\mathrm{j} \wedge op_\mathrm{i}$ and op_j conflict.

While the definition of (strict) serializability as given above only requires equivalence to *any* serial schedule, one can draw conclusions on the ordering of transactions to be fulfilled by the equivalent serial schedule from a serialization graph. Assume such a graph G, and let exist an edge from T_1 to T_2. Obviously, first T_1 executes an operation, later on T_2 performs a conflicting one. Thus, in a serial schedule (which has to order conflicting operations in the same order) T_1 has to be executed before T_2. In this case, we also say that T_1 is *serialized* before T_2. Additionally, *the serialization order* is a set of ordered pairs consisting of pairs of transaction connected by an edge in the serialization graph.

Furthermore, assume a cycle, thus (T_1, T_2) in E and (T_2, T_1) in E^*, where E^* is the transitive closure of E. Following the argumentation above, it turns out that no (equivalent) serial schedule can exist, since T_1 has to be serialized before T_2 and T_2 before T_1, which is impossible. Thus, the following theorem is a graph theoretic characterization of serializability:

Theorem 1
A schedule is serializable iff the corresponding serialization graph is acyclic.

2.2 Recovery

The notion of "recovery" comprises concepts and techniques that enable the DBMS to recover from (be resilient against) any kind of failure, be it a user or program error, an operating or runtime system failure, a disk head crash, fire or earthquake. The task of recovery is simply to reset the database to a consistent state (therefore, to recover a meaningful database state). However, even rather smart recovery mechanisms cannot protect against any conceivable kind of failure, i.e. they only can make the loss of data less likely. Furthermore, the types of failures that the DBMS should be able to deal with is a matter of cost. [Reuter 87] distinguishes the following kinds of recovery:

- R1-recovery: due to errors in the application, to deadlocks (see below), or to the violation of integrity constraints, one or more transactions can be rolled back in an isolated fashion, i.e. without affecting any other concurrently active transaction (*partial rollback*).
- R2-recovery: in case of a system crash, the consistent (hence, desired) database state is determined by the set of committed transactions. In this case it might be necessary to *redo* some of the committed transactions (*partial redo*).
- R3-recovery: in case of a system crash, the desired consistent database state is defined by the effects of committed transactions *only*, i.e. any transaction that has been active at the time of the crash has to be rolled back (*complete undo*).
- R4-recovery: while in the three cases above secondary storage (the *materialized database*) has been physically intact, a *head crash* can destroy the database *physically.* In this case, a previous database state has to be restored from an archive and all the transactions which committed since the archive copy was made have to be redone (*complete redo*).
- A further kind of recovery (not listed in [Reuter 87]) is *disaster recovery*, typically necessary in case of fire, earthquake, or nuclear bombing. In these cases, the archive may be destroyed as well, therefore a (geographically remote) backup database is required (for disaster recovery mechanisms, see e.g. [Mohan 90]).

The decision what kind of recovery has to be actually provided is mainly the problem to balance the cost of recovery and the expected cost due to losses of data in case of unrecoverable failures. For that matter, R1-recovery is a must for a DBMS, i.e. at least aborts initiated by users or by the system have to be supported. Additionally, since (although dependent on hardware configuration, operating system and the DBMS software itself) crashes usually are likely to occur several times a week, R2- and R3-recovery have to be provided as well. Not supporting these kinds of recovery would imply that users loose work of several hours several times a week. On the other hand, head crashes are rather rare, approximately in the range of years per disk. Even in the case of head crash, loss of data can be restricted to the work of one or more days by means of a dump or backup strategy. Since losses are restricted and occur rather seldom, if at all, R4-recovery is not necessary for DBMS that do not have extremely high availability requirements. The same holds for disaster recovery: since on-line backup and remote database systems are rather expensive, disaster recovery is necessary solely for systems that cannot tolerate *any* loss of data. In the following, we will omit both, R-4 as well as disaster recovery from the discussion.

2.2.1 Undo and Redo

Depending on the principles of the DBMS (page) buffer management, more or less recovery has to be done in case of transaction abort or system failure [Haerder 83]. For example, if the physical database (i.e. the database state as it is stored on non-volatile storage) always contains modifications of all committed transactions only, no recovery would be necessary at all. Although this ideal is technically possible, it might not be efficient as well (in terms of I/O costs).

Therefore, if the DBMS can materialize modifications before commit (the *Steal* option in [Haerder 83]), the DBMS must be prepared to undo these modifications if a failure occurs before the transaction can commit. Similarly, if modifications of a transaction need not necessarily get materialized (at the latest) at commit (the *Force* option in [Haerder 83]), then the system must be able to redo those modifications in the case of a crash.

In other words,

- in case the DBMS uses the *NoSteal/Force* strategy, there is neither a need for redo nor for undo,
- in case the DBMS follows *Steal,* undo information has to be written on stable storage before modification get permanent (*undo rule*, write-ahead-log principle, or WAL principle).
- in case the system uses *NoForce*, redo information has to be written before commit.

The stable storage where undo and redo information is written to is called the *log*, the process of writing redo or undo information is called *logging*, and the responsible component is the *log manager*.

2.2.2 Logging and Recovery

Logging means to store redo or undo information on stable storage, and in principle can be done in various ways. First, *state-based logging* means to save the *value* of (e.g.) pages; those pages are then called *before images* or *after images*, depending on whether they represent undo or redo information, respectively.

However, since it is in general rather expensive to store entire pages (or objects, records) whenever only some bytes were modified, the second alternative is *operation-based logging*. In this approach, redo information denotes the modification operation (or its identifier) as well as parameters supplied to the operation. Undo information is either derived from the redo record, or it is stored explicitly. In the latter case, undo information contains an *undo operation* (identifier) as well as parameters required for the execution of the undo operation.

Another question is how to organize the log. In case of restart, the DBMS first has to remove materialized modifications of uncommitted transactions in reverse order. Therefore it is useful to organize the log as a sequential file containing the log records of the modifications as they were serialized by the synchronization component. In case of a crash, the recovery manager reads the log in reverse order, analyzes it (e.g. reconstructs information on active and committed transactions) and applies undo operations for any non-committed transaction. This so-called undo pass is followed by the redo pass, which installs all non-materialized modifications of committed transactions.

However, since single transaction abort has to be possible as well (i.e. in case of user or

DBMS initiated abort), the log has to be on-line, i.e. it must be easily readable by the DBMS during normal processing. Since for transaction abort the DBMS is interested in log records of the transaction in question only, it is not appropriate to scan the log in order to find the transactions' log records, rather is it reasonable to chain the log records of each transaction. With this technique, it is easily possible to apply undo operations or to write before images for a select transaction in reverse order.

2.3 Nested Transactions

In this section, we will introduce the basic notions for nested transactions.

As already mentioned, nested transactions allow to execute transactions within other transactions. From a language point of view, this is possible in that the commands *begin_-transaction, commit*, and *abort* are allowed inside transactions again. Any transaction that is performed within another transaction is called a *subtransaction*, while the latter (the "calling" transaction) is said to be *nested*. This terminology is refined as follows:

- A transaction that is not a subtransaction (i.e. not executed within another transaction) is called *top-level* or *root* transaction.
- A transaction without subtransactions is called a *leaf transaction*.
- If a transaction executes a subtransaction, the former is said to be the *parent transaction* of the latter.
- For a subtransaction, its *ancestors* are the transitive closure of the parent relationship. *Indirect ancestors* are ancestors except the parent.
- If two subtransactions have the same parent transaction, they are said to be *siblings*.

Thus, a nested transaction, especially a root transaction with all its subtransactions and their subtransactions (and so on recursively), can be represented by a tree, the so called *transaction tree* (since the parent of a subtransaction is uniquely defined). Multiple, parallel nested transactions are then represented by a forest (*transaction forest*), i.e. a set of trees.

How do nested transactions relate to the ACID principle? Both, root as well as subtransactions, are atomic. Top level transactions are units of consistency, while subtransactions have to fulfill those consistency constraints that were valid at their start time. Root transactions are isolated from each other, as are sibling transactions. However, subtransactions are not isolated from their parent (and ancestors). In fact, a subtransaction may access objects which either were not accessed by any other active (sub)transaction at all, or were accessed by ancestors only. Thus, since root transactions are isolated from other root transactions, it follows that entire transaction trees are isolated from each other.[2] Therefore, this kind of nested transactions is called *closed nesting*, while the kind of nesting where transaction trees are not isolated from each other is termed *open nested* transactions. Finally, in order to achieve proper isolation among transaction forests and among siblings, subtransactions "inherit" the rights to perform read or write operations on specific objects to their parent at commit time. The semantics of this inheritance is that, after a subtransaction has terminated, the operations executed by the subtransaction can be regarded as executed by the parent.

2. This is the reason why inter-transaction parallelism does not increase in comparison to flat transactions.

Finally, while (modifications by) root transactions are durable, subtransactions are durable only with respect to their parent (and so on recursively). If a subtransaction commits, its effects are durable in principle (and can be observed by siblings). However, if the parent aborts later on, the subtransaction has to be rolled back as well. Thus, in essence each parent decides whether its subtransactions are durable, and in opposite to the case of flat transactions (and root transactions) this decision (i.e. first commit, later on abort) can be revised *after* the commit of the subtransaction. Clearly, when a nested transaction aborts, this decision is propagated recursively to all its descendants. Table 1 summarizes the ACID properties for flat, root, and subtransactions.

	Atomicity	Consistency	Isolation	Durability
Flat Transactions	+	+	+	+
Root Transactions	+	+	+	+
Subtransactions	+	(+)	+	(+)

Table 1: ACID Properties of Flat and Nested Transactions

Another crucial point (or design issue) is the mode in which subtransactions are executed. In principle, there are two choices: subtransactions can be executed synchronously or asynchronously [Haerder 87]. In the synchronous mode, a (nested) transaction calls a subtransaction and waits until the subtransaction terminates, i.e. does not continue execution as long as the subtransaction is running. In the asynchronous case, the nested transaction may resume execution as soon as the subtransaction started, e.g. may at once start further subtransactions. While synchronous subtransactions are easier to implement, they do not affect parallelism in comparison to flat transactions. Asynchronous subtransactions are more complicated, but support intra-transaction parallelism. Furthermore, they might be very useful e.g. in a workstation-server environment, where a user can run multiple (sub)-transactions in various windows under control of a common nested transaction.

Serializability of Nested Transactions

How can we characterize serializability for (closed) nested transactions? In the synchronous case, this is quite simple, since a nested transaction always executes at most one transaction at a time, i.e. automatically serializes its subtransactions. In the asynchronous case, due to intra-transaction parallelism, things are slightly more complex.

Intuitively, serializability for (asynchronous and closed) nested transactions can be characterized as follows: starting from the leafs of all the transaction trees, serializability has to hold on the leaf level. If two leaf transactions execute conflicting operations, they in turn conflict (i.e. are <-related). Obviously, the parent transactions have to observe the <-relation "inherited" from the children, and so on recursively up to the roots transactions.

2.4 Transaction Management in Layered Architectures

2.4.3 Levels of Synchronization

At least as a help to understand the structure of a DBMS, it is useful to assume a layered architecture [Haerder 87b], where each layer supports its own notion of "object" and operations on these objects, and is implemented by means of operations provided by the next lower level. Therefore, the question is on which of those levels to place concurrency control and recovery. For the discussion of the alternatives we assume the application of some locking technique(s), which in fact is the standard solution in database system concurrency control (and is described in a following section). Precisely, the following alternatives exist:

- page-level locking,
- object-level locking (i.e. on the level of data model "objects"),
- complex-object-level locking (if the data model provides for complex objects (e.g. [Dittrich 90]), locking can be done on complex-object-level).

In the first alternative, pages are the only units of locking. This approach not only is the traditional one, it is also used in recent developments (e.g. [Deux 91]). However, it has some major drawbacks:

- if different objects are located on the same page, a transaction may block several others, although the sets of objects to be accessed by the transactions are disjoint. Therefore this approach is only feasible under the assumption that clustering is rather smart and a transaction is likely to access all the objects located on one page (e.g. components of a complex object),
- in this approach, page locks have to be held until the termination of the top-level transaction (or abort of a subtransaction). Obviously, this leads to catastrophic performance when locks on index pages are held for such a long time. Therefore, special techniques have to be included for concurrency control of index structures (e.g. [Bayer 77]). Although this is feasible, this approach is no longer simple and modular.

Alternatively, one may apply concurrency control on logical objects or data model objects without considering complex objects. However, provided we do have complex objects, this may as well lead to incorrect database states. Assume two transactions, were the first one (directly) reads a dependent subobject, while the second one concurrently deletes the complex object containing the former one (and thus deletes the dependent subobject as well). Therefore, complex objects (if present) have to be taken into account by the concurrency control mechanism of a DBMS.

When complex objects are units of synchronization, one nevertheless cannot forget about synchronization and recovery on page-level at all. If the database system does not guarantee that DML-operations are executed atomically (e.g. in that the DBMS executes at most one DML-operation at a time), then various DML-operations belonging to different transactions may access the same page simultaneously, which occasionally may lead to problems. Therefore, synchronization on page level has to be done in addition to synchronization for (complex) object operations. Similarly, recovery has to be provided for page operations: even if we assume logical, operation-based undo, basic recovery management on page level has to ensure that the recovery manager encounters a consistent state in which undo operations are applicable at all (thus, the DBMS has to ensure that DML- as well as page operations appear to be failure atomic).

Considering the realization of a transaction subsystem for a DBMS supporting complex objects, there are the following requirements and questions:

- on one hand, we would like to synchronize on a more abstract level than pages, while on the other hand page-level synchronization and logging are necessary in order to ensure (failure as well as synchronization) atomicity of page and DML-operations. Thus, the question is how page- and object-level synchronization and logging can be reasonably integrated.
- synchronization on page level is necessary, but locks on pages should be held as shortly as possible. Similarly, recovery on page level is necessary, but on the other hand is not correct if object-level locking is used.
- therefore, if the DBMS of interest is realized on top of an object server rather than from scratch, the question (or design issue) is how such an object server could be integrated with “the rest” (upper layers) of the DBMS, especially, from the viewpoint of an object server, what should be provided, or, from the viewpoint of the upper layers, what can be assumed to be supported by the server.

All these issues are addressed by *multi-level transaction management,* which is additionally rather elegant as well as efficient. Multi-level transaction management is well-suited in layered architectures [Haerder 87b], and was implemented in DASDBS [Weikum 84] and similarly in System R [Gray 81]; a more formal discussion is given in [Weikum 91] and with similar results in [Moss 86b]. The idea behind multi-level transaction management is to regard DML-operations as open nested transactions again that execute (unlike “normal” transactions not the usual DML-operations, but) operations of the next or any other lower level, e.g. operations on physical objects or on pages.

Multi-level transaction management works as follows, assuming a two-level scheme: the top level (L2) is formed by the user-visible transactions, i.e. the leaf transactions of the closed nested transactions executed on behalf of users or applications. These transactions are implemented as sequences (or partial orderings) of DML-operations like retrieval or update of data model objects. Additionally, this (data model) layer contains a scheduler that synchronizes DML-operations.

Additionally, each DML-operation (i.e. L1-action) in turn is executed as a transaction. In a two level scheme, such a transaction is implemented by page operations (i.e. by L0-operations). The layer providing page operations again runs its own scheduler that controls concurrent DML-operations. Thus, a DML-operation has to acquire page locks during its execution. Since multi-level transactions are open nested ones, page locks are released at the end of the DML-operation. Figure 1 shows an example. T_1 and T_2 are leaf transactions that read and/or write two objects *x* and *y*, respectively. Unfortunately, both objects are

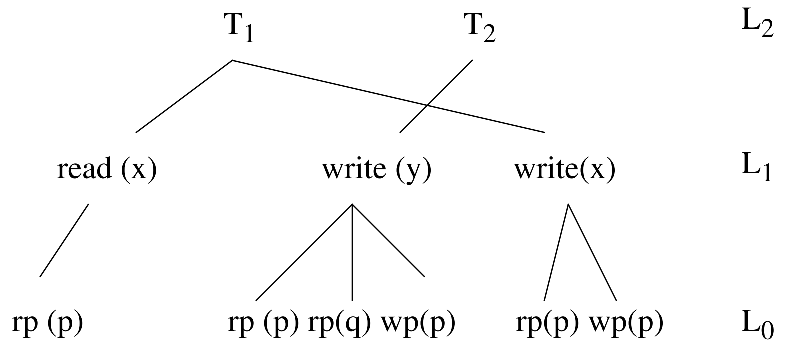

Figure 1. Multi-level schedule

stored on the same page, thus, using page level locking, T_2 would be blocked until the end of T_1 or the end of the root transaction containing T_1. However, if a two-level transaction management is employed, the read lock on page *p* is released at the end of the *read(x)* operation.

2.4.4 Serializability of Multi-Level Transactions

Although intuitively the two operations above do not violate serializability on the level of DML-objects, they do so on the page level. Thus, one might question whether serializability really holds in this case. For that matter, we give a formal justification for this approach.

We assume that page operations are (synchronization) atomic, i.e. multiple page operations are themselves not executed in an interleaving manner. Then, let an *n*-level schedule be an *n*+2-tuple (A_n,...., A_0, $precedes_0$), where

- A_n is a set of (leaf) transactions,
- $A_i = \{a_{j1},..., a_{jkj} \mid a_j \in A_{i+1}$ and $k_j \in \mathrm{N}\}$ is a set of L_i actions,
- $precedes_0 \subseteq A_0 \times A_0$ is a partial order on L_0-*actions.*

$Precedes_i$ reflects the intratransaction precedences, i.e. represents the order in which (sub) transactions or actions were invoked. Additionally, some of the operations invoked on a given level may conflict (in the usual sense of conflict); this circumstance is reflected by layer-specific conflict relations CON_i, where an action pair $(a, b) \in CON_0$ implies that either $a\ precedes_0\ b$ or vice versa. For a given level, we assume that, if operations on level *i* conflict, there has to be a conflict between operations spawned by the two operations on level *i*-1. Given an *n*-level schedule, we define quasi-orders $<_i$ as follows:

1. for $i = 0$, $<_0 = precedes_0 \cap CON_0$,
2. for $0 < i \leq n$: $a <_i b$ iff *a* executes *a*', *b* executes *b*', such that $CON_{i-1}(a', b')$ and $a' <_{i-1} b'$.

Clearly, such a quasi-order needs not be acyclic, hence the term "quasi". However, if $<_i$ is acyclic for each *i*, then the corresponding multi-level schedule is said to be *n-level serializable*.

Additionally, a theorem states that a schedule is *n*-level serializable if all level-by-level schedules (i.e. schedules that consider two levels L_i and L_{i-1} only) are serializable (as defined in a previous chapter). This theorem is rather useful, since now we can employ the usual concurrency control techniques on each layer, with each of the corresponding schedulers serializing "its" transactions in terms of conflicts of the next lower level, i.e. none of the schedulers neither needs to know what "happens" above it or more than one layer below it. Additionally, we do not need to have a scheduler on each level, in principle one scheduler on the page level and one for higher-level objects suffices.

The concept of multi-level transactions enables us to cope with layered architectures of DBMS in a rather elegant way. They are also useful if the DBMS is implemented on top of an *object server* [Geppert 92], which provides for general functionality like page buffer management, indexes, and storage object management, while the application-specific functionality (e.g. complex data model objects, query processing and optimization) is implemented by the "upper layers" on top of the server. In this case, multi-level transactions allow us to subdivide the overall, complex task of transaction management into subtasks and to assign those to the server and the upper components, respectively. Since DML-operations are open nested transactions, page locks can be released after DML-actions, thus acceptable parallelism even on indexes (i.e. on index pages) results (and therefore a special concurrency control mechanism for indexes like in [Bayer 77] is not necessary).

2.4.5 Recovery for Multi-Level Transactions

In this subsection, we elaborate on recovery concepts for multi-level transactions. For the schedule shown in Figure 1, assume that one of the transactions issuing the two DML-operations aborts later on. Now, assume state-based logging, i.e. during rollback e.g. a before image page is installed. Obviously, the rollback will presumably corrupt the modifications of the other DML-operation. This is due to the fact that for state-based locking the first operation released its locks "too early", i.e. at a point where it was not clear whether the transaction having issued the DML-operation will abort or commit. Therefore, state-based locking is not applicable, but operation-based logging (i.e. recovery by means of compensating actions) is. Concretely, a (successful) *insert* operation can be rolled back by means of a corresponding *delete* operation, and so on. Thus, for any modification operation on page and DML-level, a compensating action has to be specified, where for page *writes* the compensating action is a *write* of the old state (before image).

In general, each layer (at least those with a scheduler of their own) provides for its own logging and recovery, and thus manages failure atomicity of its transactions. In the case of abort of an action on level L_i, the corresponding recovery manager consults its log and executes appropriate undo actions (just like in the normal case).

In the case of a crash, things are more complex. In general, each layer is obliged to enforce atomicity and durability of its transactions. Thus, from the bottom (i.e. starting on the page level) up to the level of "user" transactions, each layer rolls back its active transactions. As an example, assume a crash during the second write operation in the example above. Then the page level recovery manager is responsible to install a state consistent on the page level as it had been when the *write* operation began its execution. Likewise, other active page-level transactions have to be reset. After a consistent state is reinstalled on the page level (which includes the effects of committed transactions, e.g. *writes* prior to the *write(x)* operation), the recovery manager on the next higher level starts to undo *its* active transactions, i.e. other *writes* or modifications that occurred before *write(x)*.

3 Techniques

Up to now, semantics and concepts for transaction management have been described. In this section, we focus on techniques and algorithms realizing those concepts.

3.1 Architecture

A transaction subsystem is a part of a DBMS that provides for the management of multiple, concurrently execution transactions; it in turn consists of various components. Figure 2 shows a possible subsystem architecture. Besides the *transaction manager* (TM), a *scheduler* (lock manager, LM) exists and ensures that only (strictly) serializable schedules are executed. Furthermore, the *log manager* (LogM) is responsible to record information (in the *log*) which lateron is used when transactions are aborted or rolled back. Abort in turn runs under control of the *recovery manager* (RM), which reads the log and executes appropriate actions. Additionally, without loss of generality we assume that the DBMS uses an object server as underlying component for the management of pages, the page buffer, in-

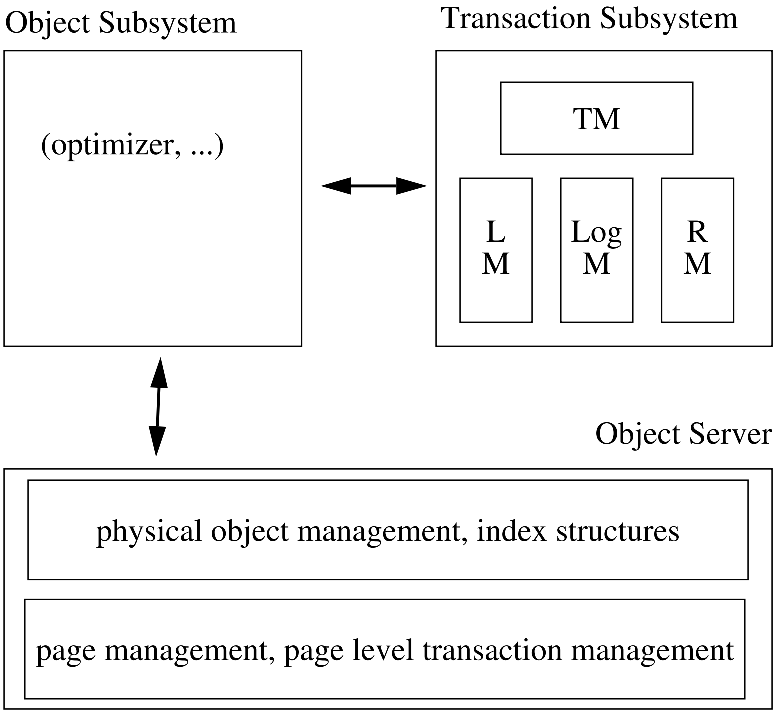

Figure 2. Subsystem DBMS Architecture

dexes, and (physical) storage objects.

3.2 The Transaction Manager

The transaction manager (TM) is invoked whenever transactions (top-level or subtransactions) start, commit, or abort. When a transaction starts, TM assigns a *transaction identifier* (*tid*) to it, writes a BOT-record to the log, and initializes further data structures (e.g. records the parent transaction and initializes a list to store *tid*'s of subtransactions). Especially, information on the parent is required rather often, thus it may be too expensive to walk through a tree every time it is to be checked whether a transaction is an ancestor of another one. A cheaper solution would be to encode information on ancestors in the *tid*. For example, root transactions are assigned unique *tid*'s, and the *tid* of a subtransaction is built from the parent's *tid* and a number unique within the parent's subtransactions. In this case, a (nested) transaction is an ancestor of a subtransaction if the *tid* of the former is a prefix of the *tid* of the latter. Since *tids* are reasonably required to be of fixed length, the nesting depth of transactions is restricted.

3.3 The Scheduler

The *scheduler* is a component of the transaction subsystem that ensures that only serializable schedules occur as results of interleaved executions of transactions. For that matter, it receives requests from transactions and validates whether granting those requests violates serializability or not. A request usually is a triple (*tid*, *ob*, *op*), where the first component is a transaction identifier, the second is an object and the last one is an operation to be performed on the object. In principle, requests for sets of objects (and sets of operations) are also possible. If the request can be granted (i.e. does not hurt serializability), the scheduler invokes the *object manager* to perform the desired operation. If not, it may delay the request or abort the requesting transaction or another conflicting transaction.

Since a lot of techniques for synchronization have been proposed, it is obvious to ask for the "best" one. Therefore, one may ask what a "good" scheduler actually is, which in turn depends on the characteristics of the expected transaction profile in terms of the following criteria:

1. *Duration*: transactions can be short or of long duration like in design applications, e.g. can potentially span multiple sessions.
2. *Transaction semantics*: precise knowledge about objects that will be accessed by any transaction can exist or derived from transactions programs, or those knowledge in general is not available.
3. *Conflict rates*: transactions can be likely to conflict, or conflicts between transactions can be rather seldom (i.e. a high percentage of read-only transactions like in statistical applications).

We assume that transactions are short, that in general knowledge of objects to be accessed does not exist, and that transactions are likely to conflict. Thus, in the light of those characteristics and apart from allowing for serializable schedules only, useful synchronization techniques *balance* the following requirements:

- They provide for as much parallelism as possible,
- They require as little overhead as possible, both in terms of space and time.

Database technology has developed a variety of scheduling techniques, which can be classified according to the time when the operations on objects on behalf of a transaction are actually validated, i.e. tested for conflict with operations of other transactions (the *validation time*):

- conservative techniques (*preclaiming*) require a transaction to specify the set of objects it wishes to access *at transaction start* time. Thus, conflicts can be detected as soon as a transaction starts.
- pessimistic techniques assume that conflicts are likely to occur. Consequently, those techniques validate before operation execution, i.e. whenever an operation has to be executed on behalf of a specific transaction, they check whether that operation conflicts with any (recently executed) operation of another transaction. As mentioned above, the scheduler may defer the desired operation, thus it *reorders* arbitrarily (i.e non-serializable) sequences of operations such that the resulting schedule is serializable.
- optimistic techniques assume conflicts to happen rather rarely (if at all); they permit everything and check at the end of a transaction whether its operations conflict with other transactions. In the case of conflict, the transaction may not commit and has to be rolled back.

Since the precise set of objects to be accessed usually is not known beforehand, parallelism is rather low in the conservative approach. Furthermore, while the optimistic approach allows for higher parallelism, the optimistic assumption in general is not justified. Therefore, although pessimistic techniques imply higher overhead and are suboptimal in terms of parallelism, they are the standard synchronization techniques in database systems. Among pessimistic techniques, strict two-phase locking (see below) is the most widespread one.

3.4 Synchronisation Techniques

In this subsection, we focus on techniques and components for synchronization. Recovery techniques are described in a later section.

3.4.1 Two-Phase Locking

In the locking approach, a *lock mode* is defined for each kind of operation (e.g. *read* and *write locks* for *read* and *write* operations, respectively). Before a *read* or *write* operation can be performed on an object, an appropriate lock has to be acquired. Furthermore, a *lock compatibility function* is defined, where requested locks are compatible with existing ones if the corresponding operations commute, and are incompatible, if the related operations conflict. If locks are incompatible, they are also said to *conflict*. In order to ensure strict serializability, a transaction may not acquire locks after it has released some lock, hence the term two-phase locking (2PL); additionally, all the locks have to released in an atomic way at the end of a transaction (strict 2PL).

The principles of synchronizing active transactions by means of locking are described in the following *protocol*:

1. Whenever a transaction requests to execute an operation *op* on object *x*, it has to obtain a lock on *x* in the appropriate mode.
2. If no conflicting locks are held by transactions, or only by the requesting transaction itself, the lock can be granted. If any other transaction already holds a lock in a conflicting mode, the (newly) requested lock cannot be granted. In this case, the requesting transaction is either delayed (blocked) or aborted.
3. All the locks a transaction acquired during its execution are released in an atomic manner at its termination (be it *commit* or *abort*).

For nested transactions, this protocol is modified as follows:

1. Whenever a transaction requests to execute an operation *op* on object *x*, it has to obtain a lock on *x* in the appropriate mode.
2. If all conflicting locks (if any) are held by ancestor transactions or by the requesting transaction itself, the lock can be granted. If any other transaction already holds a lock in a conflicting mode, the (newly) requested lock cannot be granted. In this case, the requesting transaction is either delayed (blocked) or aborted.
3. All the locks a subtransaction acquired during its execution are inherited to its parent at *commit*.
4. All the locks a (sub) transaction acquired during its execution are released in an atomic manner at *abort*.
5. All the locks a top-level transaction acquired during its execution are released at its termination (be it *commit* or *abort*).

3.4.2 Intention, Implicit, and Multigranularity Locking

In some sense, the locking protocols sketched above are not sufficient, e.g. in cases when multiple objects related by some specific relationship have to be locked. As an example, assume one transaction is modifying the definition of an object type, while a second one tries to insert an object of that type. Clearly, inconsistency is likely to happen in this case. Since type definitions are represented by objects as well, they too can be locked. Additionally, all existing objects of the respective type could be locked, which, although possible, is regarded as too expensive.

Additionally, in our scenario, insertion of new objects cannot be prevented. Assume one transaction starts a scan on an object extension while a second one tries to insert a new object. In that case, the scan might not be affected by the insertion; if the inserting transaction commits and later on the first transaction queries the cardinality of the scanned extension, this cardinality does not equal the number of actually read objects (*phantom problem*).

Those abnormal situations can be prevented by *implicit* and *intention locking*, which take different granularities and relationships among them into account. In the simple case, intention locking is defined for hierarchies of locking units, e.g. databases, which consist of extensions, which in turn contain objects.

Implicit locking means that, whenever a lock (in any mode, like the usual *read* and *write* locks) is set for a given granule, some other granules are viewed as if they were locked as well. For example, if the entire database or a logical database [Dittrich 90] is locked in *read* mode, all extensions and all instances of those are regarded as locked in the same (*read*) mode. The reason why those locks are termed “implicit” stems from the fact that they do not exist physically, but are derived from the locks of other (coarser) granules. Thus, if a transaction modifies an object type definition, the members of the corresponding extension are locked in *write* mode as well; therefore, none of the instances can be read or written by concurrent transactions. Additionally, if the extension itself is locked, no new objects can be created while the definition is modified.

Implicit locking can be used for *lock escalation*. Although usually it is desired to lock as few and as fine granules as possible (i.e. single objects instead of the entire database), this may blow up the lock table. In those situations it is reasonable to convert a set of locks of (a homogeneous set of) granules into *one* lock of the next coarser granule. For instance, if a transaction scans an object type extension, the scheduler may start to lock the entire extension when a given number of locks of instances is reached.

Intention locks are useful if we desire to express that a transaction *intends* to access a granule (or one or several of its instances) in a specific mode. Therefore, the lock modes so far can be used for any granule. Additionally, two new lock modes are introduced for *read* and *write*: *intention read* means that the holder intends to read some part of the locked unit, i.e. if an extension is locked in *intention read* mode, then the transaction desires to read some instance of the extension. The same holds for *intention write*. Finally, a lock mode *read intention write* is provided that allows the holder to read a granule and to reserve the right to write an instance of it (i.e. reading a type definition and writing an instance of it).

Finally, the compatibility of the (enhanced set of) lock modes as used e.g. by System R [Gray 81] is shown in Table 2. While the rows correspond to existing locks, the columns

	r	w	ir	iw	riw
r	y	n	y	n	n
w	n	n	n	n	n
ir	y	n	y	y	y
iw	n	n	y	y	n
riw	n	n	y	n	n

Table 2: Compatibility Function of Intention Locking

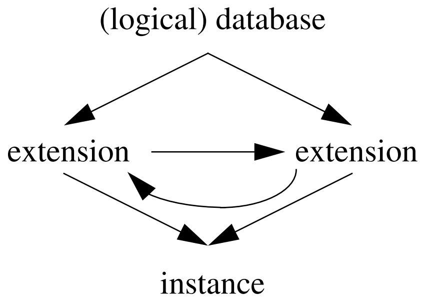

Figure 3. Locking granularity graph

represent requested locks.

A still open question are the *protocols* than can be used for intention and implicit locking. For the first case, such a protocol is defined for databases, extensions, and instances. If inheritance and substitutability are supported by the data model such that an object may be instance of multiple extensions, a *locking granularity graph* (essentially, a rooted graph) as depicted in Figure 3 can result. In this figure, an arrow has the semantics of "contains" or "consists_of".

Intention and implicit locking will not work correctly if transactions start to lock in the “middle” of a graph. Rather, the following protocol has to be obeyed:

1. If a non-leaf node is locked in *r* or *w* mode, all descendants are implicitly locked in *r* or *w* mode, respectively. If a non-leaf node is locked in *riw* mode, all descendants are implicitly locked in *r* mode, and some descendants may be locked explicitly in *w*, *iw*, or *riw* mode.
2. Before requesting an *r* or *ir* lock on a node, all ancestors have to be locked in *ir* or *iw* mode.
3. Before requesting an *w* or *iw* lock on a node, all ancestor nodes have to be locked in *riw* or *iw* mode.
4. Locks are released either at the end of a transaction (in either order, but atomically), or in leaf to root order.

Thus, while locks are acquired from the root to the leaves, they can be released in reverse order (but in an atomic fashion).

That protocol is still fair as far as readers and writers are compared, however, fairness might be weakened for the sake of decreasing locking overhead. In the protocol above, readers need *ir*, *r*, or *riw* locks on all parents. This is called the *total parent rule* in [Korth 83]. Other possibilities include

- the *majority parent rule*: let *L* be a lock mode (e.g. *r*) and *x* be a non-root node in the lock graph. A transaction observes the majority parent rule with respect to *L*, if it can acquire *L* or *iL* only if a majority of the parents of *n* is already locked in *iL* mode by the transaction.
- the *single parent rule*: let *L* be a lock mode (e.g. *r*) and *x* be a non-root node in the lock graph. A transaction observes the single parent rule with respect to *L*, if it can acquire *L* or *iL* only if at least one of the parents of *n* is already locked in *iL* mode by the transaction.

One can use the total parent rule for *r* and *w* locks, or single parent rule for *r* and total parent rule for *w*, or majority parent rule for both, respectively.

Locking Type Lattices and Complex Object Hierarchies

So far, we have assumed that single objects and extensions are the units of locking. However, pathological situations may occur due to the presence of type lattices and is-part-of graphs. For the first case, assume a transaction that locks a type, and afterwards another transaction changes the definition of a supertype. Presumably, this modification would corrupt the view of the subtype. The situation is even worse when multiple inheritance is possible, i.e. an object type may have more than one supertype, and thus several predecessors in the lock graph [Garza 88]. In this case, implicit locking is not sufficient. Therefore, [Garza 88] proposes to *explicitly* lock any subclass with more than one superclass. Additionally, intention locks along any ancestor chain are required for those classes (which is rather expensive for classes near the leaves of the graph).

The situation is likewise complicated for complex objects, at least when direct access to subobjects, i.e. not (only) via the root of the complex object graph, is possible. For instance, assume a transaction reading a complex object while concurrently another transaction deletes one of the (direct or indirect) component objects. Similarly, let a transaction directly access a subobject, while another transaction tries to delete the root. Consequently, complex objects have to be units of locking as well, i.e. have to be included in the granularity graph.

A possible solution is described in [Garza 88]. Three new (intention) lock modes (*iro, iwo, riwo*) are introduced. Whenever one type is locked in any of the five (intention) lock modes, any component type is locked in one of the three modes above. The compatibility matrix of this enhanced set of lock modes is shown in Table 3. Using this protocol allows multiple

	ir	iw	r	riw	w	iro	ixo	riwo
ir	y	y	y	y	n	y	n	n
iw	y	y	n	n	n	n	n	n
r	y	n	y	n	n	y	n	n
riw	y	n	n	n	n	n	n	n
w	n	n	n	n	n	n	n	n
iro	y	n	y	n	n	y	y	y
iwo	n	n	n	n	n	y	y	n
riwo	n	n	n	n	n	y	n	n

Table 3: Compatibility Matrix of ORION

transactions to read or update components of different root objects. However, one transaction reading a complex object blocks any other directly writing transaction (even if the object to be written is not subobject of the one accessed by the first transaction), as a directly reading transaction blocks transactions that modify subobjects via the complex object graph.

From a performance point of view, this approach favors efficiency of lock management over parallelism.

A higher degree of parallelism can be achieved if identifiers of root objects are known for each (direct or indirect) subobject (note that there may be several roots, since subobject potentially can be shared). In this case, when a transaction directly accesses an object, it has to be checked whether it is a subobject, and if yes, the roots have to be locked in an appropriate mode.

The database system DAMOKLES [Abramowicz 89] followed this approach for *checkout* operations on structured objects. However, either root identifiers have to be stored at each (in)direct component, or a special table for subobject relationships has to be maintained. Besides the space overhead, an extra price is paid when the structure of a complex object is changed, since links to the root objects have to be modified as well. Nevertheless, this solution probably balances overhead and parallelism in a better way.

Finally, another possibility is proposed in [Herrmann 89] for disjoint, non-recursive objects and in [Herrmann 91] for arbitrary structured objects. For ease of exposition, we start with the first approach. Initially, all objects and values are classified into *basic lockable units* (BLUs, “flat” objects, or tuple attributes), *homogeneous lockable units* (HOLUs, sets or lists), and *heterogeneous lockable units* (HELUs, complex objects). Now, the idea is not to have a fixed lock granularity graph, but to tailor such a graph for each query against a complex object type. Given a query accessing complex values or objects, an estimation on the selectivity of the query is made. If the complex value or object is a HELU and the selectivity of the query exceeds some given limit, then the HELU is inserted into the lock graph; the same holds for HOLUs (potentially with different selectivity limits). If a HOLU or HELU is a node in the lock graph, (appropriate) locks on the HOLU (HELU) have to be acquired instead of on the single elements (this way one tries to avoid lock escalation). This process results in the *object-specific lock graph*. Starting with this graph, a query-specific lock graph is constructed, which additionally contains information on the lock modes that have to be acquired for the various nodes of the graph.

The case of recursive objects (i.e. subobjects of the same type as the complex object itself) is reflected in that BLUs in the respective lock graphs represent references to objects of a higher nesting depth. While “superobjects”, i.e. objects containing an object in question as a (direct or indirect) subobject have to be locked in intention mode, subobjects are locked implicitly in the same mode, i.e. locks of (recursive) complex objects are inherited to the subobjects. Note that the intention locking of superobjects implies both, more space (storage of references to superobjects) as well as more time (each superobject has to be locked in intention mode).

In the case of shared subobjects, locks have to be propagated “downwards”, i.e. to the shared subobjects (more precisely: to all subobjects that could be shared). However, in order to admit those lock requests, the predecessors of the sharable subobjects have to be locked in corresponding intention modes (“upward propagation”).

Although this approach tries to balance locking overhead (i.e. to anticipate lock escalation) and the degree of parallelism, the efforts necessary to construct lock graphs is far from being negligible. Additionally, the benefit of the approach strongly depends on the quality of selectivity estimates.

3.4.3 Weaknesses of Two-Phase Locking

In this subsection, we describe some weaknesses and problems of 2PL as well as potential solutions.

Parallelism

Parallelism is rather low when strict two-phase locking is used; it implements a proper subset of strictly serializable schedules, i.e. not all schedules permitted by strict serializability can be obtained by two-phase locking. While strict serializability allows an object to be read by transaction t_i and afterwards (but before the termination of t_i) to be written by another transaction t_j. Following the locking protocol as sketched above, this is not allowed by strict 2PL, rather the second transaction would be blocked until the reading transaction terminates.

Deadlocks

Usually, a transaction that requests a lock is blocked if a conflicting lock is held by another transaction; in this case we say the former *is waiting* for the latter. However, the waiting transaction (say, T) can already have acquired locks. A transaction for which another transaction T is waiting can continue execution and request further locks, some of which may be held by T. The result is that none of the two transactions can continue its execution, since they are waiting for each other in a cycle. This phenomenon is called *deadlock*, and the DBS either has to avoid such situations or to react to them. Usually, the *waits-for* relationship among transactions is represented as a directed graph (the *waits-for graph*). Thus, transactions deadlock with others if (and only if) they participate in a cycle in the *waits-for* graph.

In principle, the following strategies can be used in order to deal with deadlocks:

1. Deadlocks can be *prevented* by the synchronization technique. Conservative techniques, for instance, are proven to be deadlock avoiding (in that locks have to be acquired at transaction begin).
2. Deadlocks can be *avoided*. This strategy is possible for 2PL and allows transactions to wait for others only if no deadlock occurs, i.e. each time a lock cannot be granted it is checked whether a deadlock exists or not [e.g. Agrawal 83].
   Another possibility to avoid deadlocks is to decide whether T might wait for another transaction (e.g. T') at all. In this case, priorities or timestamps might be used, and one of the following strategies can be used:
   - **Wound-Wait**. If T is older (or has higher priority), it *wounds* T', causing the latter to abort. If T is younger, then T waits.
   - **Wait-Die**. T *waits* if it is older and *dies* otherwise, i.e. aborts itself.

   Using either wound-wait or wait-die (but not both!) guarantees deadlock freedom (i.e. absence of cycles in the waits-for graph) if unique priorities or time-stamps are used.
3. If deadlocks are neither prevented nor avoided, they have to be *detected*. Following this strategy, it is checked for deadlocks either periodically or there is a time-out whenever transactions have been idle for a specific period of time.
   Since the relevant information on waiting transactions can be represented as a graph and deadlocks occur if (and only if) there is a cycle in the graph, graph-theoretic algorithms can be used to detect deadlocks (see e.g. [Aho 75]). If a deadlock is detected, (at least) one of the involved transactions has to be aborted (the so called “victim”) and has

to be restarted. The criteria which transaction to select as a victim are another design issue; in principle it should be avoided to abort “old” transactions, i.e. transactions that already performed a lot of work that would get lost due to abort. Furthermore, when transactions are restarted, cyclic restarts (i.e. a transaction repeatedly runs into deadlock, is restarted, deadlocks again, and so on) should be avoided, too.

Lifelocks

A similarly “ugly” situation can happen due to the scheduling strategy. Assume a transaction, T_1, holds a read lock on a data item, while another transaction T_2 waits for a write lock. Now, other transactions may request further read locks on the same object (or any other non-conflicting locks). In principle, these locks can be granted, with the consequence that the “readers” overtake the transaction waiting for the write lock. In this scenario it may happen that T_2 can never acquire the write lock, since there is always a reader holding a read lock. This phenomenon is called *lifelock* or *starvation.*

Thus, another design issue is to find an adequate compromise between optimal system throughput and fairness [Agrawal 83].

3.5 Recovery Techniques

Recovery techniques for multi-level transaction management are discussed in [Weikum 90]. In this subsection, we summarize the techniques proposed there.

For page level (L_0), the use of the DB-Cache method [Elhardt 84] is proposed. In this approach, the two major components are the *cache* (the page buffer), and the *safe*, which is part of the non-volatile storage and is approximately as large as the cache. Whenever a transaction[3] (i.e. DML-action) reads a page p, it is read from the database (DB), declared as original (p_o), and marked as unchanged and fixed. If the transaction wants to change p, the page is read from the DB and declared as a copy p_c of p. If p_o is already in the cache, p_o becomes the copy p_c. In order to abort a transaction, all copy pages p_c are simply deleted and originals unfixed. For commit, all the copy pages are sequentially written to the safe. In the cache, copy pages become original pages and are marked as unfixed and changed. Changed pages may be swapped out to the DB, but not copy pages. Therefore, the physical DB always reflects an (action) consistent state. It is assumed that the cache can hold the copies of all pages accessed by active transactions.

Later on, if another transaction wants to update p, the page may be found in the cache (if it has not been flushed to the DB). Thus, p_0 is found in state changed or unchanged. Anyway, an identical copy p_c of p_o is made; if the second transaction commits, its copy p_c replaces the old original p_o.

In case of restart after a crash, pages are read from the safe according to their original write sequence; hereby newer pages replace old ones in the cache. Finally, all pages in the cache are marked as changed and unfixed.

This approach is used in [Weikum 90] for atomicity and durability of page-level transactions. However, forcing pages to the safe at commit of a DML-action seems rather costly in terms of log-I/O. Alternatively, pages may be written to the safe after commit of page-level transactions, if the following rules are obeyed:

3. Please do not be confused by the usage of the term “transaction”. For the rest of this section, “transaction” without further qualification is meant as L_0-transaction, i.e. as implementation of a DML-action.

1. Copy pages modified by a page-level transaction have to be written to the safe together (and atomically).
2. At the latest at commit of a DML-transaction, the after images, i.e. the copy pages in the above terminology, of its page-level subtransactions have to be written to the safe.
3. If the write sets of two page level transactions overlap, their copy pages have to be written to the safe together.
4. When a dirty page (i.e. a page that has not been written to the safe) is replaced (and flushed to the DB), then the copy pages of the latest page-level transaction that modified this page have to be written to the safe.

Note that this way a page-level transaction may not be durable (since we do not force copy pages to the safe at each commit). This is not harmful since essentially we are less interested in durability of page-level transactions, but in permanence of transactions on the DML-level.

For the layer above (i.e. the DML-transactions), the only rule is the write-ahead-log rule (WAL): before the effects of a DML-action become permanent, undo information has to be written to the L_1 log. Note that redo in case of a crash is not necessary on the L_1-level since it is provided by L_0.

Processing of Transaction Abort

The abortion of page-level transactions (e.g. in case of deadlock) is rather simple: as described above, the copy pages of the transactions in question have to be discarded.

If a DML-transaction (say, *T*) is to be aborted, we assume that no subtransaction of it (i.e. a page-level transaction) is active, otherwise that one will be aborted first. Afterwards, we find an action-consistent state, and the L_1 undo actions can be performed. For that matter, the L_1 recovery manager retrieves the log record written most recently by *T*. This log record is either the corresponding *begin_transaction* record, or it represents an undo action. In the first case, abort is finished, while in the second case, the log record is interpreted and corresponding (undo) actions are performed. Also in the second case, one component references the preceeding log record, which can be read, interpreted, and so on transitively.

Abort occurs concurrently to other active transactions. Therefore, depending on the set of used lock modes, the following ugly situation may happen: assume *T* is selected as a victim in case of a deadlock. Then, *T* has to undo its actions. From the original action execution, *T* already holds locks, and it presumably will require locks in order to be able to perform undo actions. If the locks required for undo are stronger, then *T* will have to upgrade those locks, which, however, may conflict with other active transactions. In other words, during the process of abort, *T* may deadlock again. Thus, the original deadlock cannot be resolved. In consequence, while those situations cannot occur with the set of lock modes selected above, one should be cautious in this point and properly regard the interaction of lock modes of action and undo actions.

Restart Processing

Warmstart, i.e. crash recovery is performed on the levels L_0 and L_1. First, pages are read from the safe and written to the cache (and perhaps to the DB). This is a single, L_0 redo pass. After that pass the DB state is (DML) action-consistent.

Afterwards, the L_1 recovery manager scans its log in reverse order. In contrast to transaction abort, the recovery manager reads all log records. If a log record belongs to a transaction whose top-level transaction had not written a *commit* record, then an undo action corresponding to the log record is performed.

Since page-level transactions need not necessarily be permanent, it may happen that a log record for a DML-action is in the L_1 log but the changes to be undone are not reflected in the database. For example, assume an operation of a list value that inserted a new member at the beginning of the list. The undo action in this case would be to remove the first element of the list. However, if after the L_0 redo pass the old list value is installed, performing the undo action is not correct. There are two ways how to circumvent this undesired effects: one is to apply idempotent undo actions only. In this case, applying the undo action more than once yields the same effect as performing undo exactly once. Thus, if modifications are not reflected after redo, one can imagine that undo already has been performed, and applying undo once again results in the desired state. The second possibility is to include corresponding page-level transaction identifiers in the L_1 log records. Then, after redo the L_1 recovery manager is informed on the set of page-level transactions that were redone and therefore can decide which undo actions are not necessary.

Due to Murphy's Law, it is rather likely that a crash occurs during transaction abort or during crash recovery itself. What would happen in such situations? Following [Weikum 90], logging is done during both, transaction abort as well as crash recovery. Thus, the (second) crash recovery would first undo undos and then undo original actions. Intuitively, this is a correct approach, however inefficient, i.e. it takes a longer time until the DBMS is again available for normal processing. If the time between failure and start of normal processing has to be as short as possible, which is the case in applications with high availability requirements (which we assume is not the case for CoOMS), an approach like in [Mohan 90] may be followed. There, log records do not necessarily point to their immediate predecessor (according to their write sequence), but to the next log record relevant for undo or redo processing.

4 Transaction Management in CoOMS

Having surveyed general concepts and techniques in the two previous sections, we now summarize the features of transaction management proposed for CoOMS.

Applications and users execute closed nested transactions. Subtransactions are executed in a synchronous mode, while the asynchronous mode would be easily possible as well with the proposed techniques. Furthermore, it is assumed that applications run on a client (workstation) and the DBMS process runs on a server. It is still an open question whether the server executes closed nested transactions or flat transactions only (see next section).

Synchronization is performed on DML-level whereby the strict two-phase locking protocol for closed nested transactions is used. This protocol also takes type lattices and complex object hierarchies into account. In other words, complex objects are the units of locking, and readers or writers via a complex object hierarchy are synchronized with direct accesses to instances.

Regarding the protocols described in the section “Locking Type Lattices and Complex Object Hierarchies” on page 21, the question is which of them to implement in CoOMS. The criterion for that decision is that the selected protocol should balance parallelism (the more, the better) and transaction management overhead (the less overhead, the better). Therefore, intention and implicit locking on type lattices is necessary, thus the ORION solution should be employed for that matter. For complex object hierarchies (and graphs), the ORION solution appears to be suboptimal in that the degree of allowed parallelism is rather low, while [Herrmann 91] allows for better parallelism, a smart lock escalation mechanism, but implies rather high overhead. Especially, since complex objects may be rather “large” (in terms of direct and indirect subobjects), it should be avoided to traverse (large parts of) a complex object graph merely for locking purposes. Especially, this is a problem when complex objects are recursive.

Therefore, we suggest a modification of [Herrmann 91] for recursive objects, which leads to lower parallelism, but is expected to decrease overhead: given that each subobject of a recursive complex object references the root of the recursive complex object graph, that root can be locked, thus implicitly locking the entire subgraph.

Recovery on DML-level is based on operation logging, i.e. for each modifying DML-operation the corresponding compensating action is logged. *Delete* operations, especially on complex objects, can be deferred for the sake of (logging) performance. Redo recovery is not performed by the DML-level, but by the lower components.

Internally it is assumed that CoOMS is implemented on top of an object server. This object server is expected to provide for primitive, page-level transaction management. The object server uses a strict two-phase locking protocol on page level and the DB-Cache method of [Elhardt 84], and thus provides for failure and synchronization atomicity and durability of page-level transactions. DML-actions are implemented as object-server-level transactions. Therefore, multiple DML-actions can execute concurrently without threatening consistency and atomicity of DML-actions.

5 More Architectural Issues

In this section, we focus on some special issues, namely transaction management in a server/client environment, and relationships of the CoOMS transaction subsystem to other DBMS components.

5.1 Relationship to Consistency and Query Optimization

A special point that should be paid attention to is the relationship of transaction management to integrity maintenance and query optimization.

Transactions (and subtransactions) are usually defined as the units of consistency. Therefore, while a transaction can install an inconsistent database state during its execution, at the latest at commit there must not be inconsistencies due to the effects of the committing transaction, otherwise the transaction is not allowed to commit. However, (implicit) integrity constraint checking need not necessarily occur at commit, specific integrity constraints (e.g. uniqueness conditions) can also be checked before commit.

While from the point of view of transaction management one alternative is as good as the other, a query optimizer may make assumptions about the time of constraint checking. For instance, assume the following query:

```
select a.name from a in assistant, b in assistant
    where a.name = b.name and not (a = b)
```

that is, select the name of all assistants that have a colleague with the same name. Now, let the `name` be constrained to be unique. Then, a smart query optimizer may deduce that the result of this query is empty, since the cardinality of queries selecting the value of unique attributes is never greater than 1. However, if integrity constraints are checked at commit, then the above assumption is no longer valid.

In other words, a decision has to be made when integrity constraints are enforced, and therefore which assumptions the query optimizer (and other components) can make about this issue.

5.2 Transaction Management in a Client/Server Environment

CoOMS is assumed to operate in a server/client environment, where the DBMS (processes) runs on a server while applications run on workstations (clients) [Dumm 91]. Therefore, one may ask how transaction management described so far fits into such an architecture, or whether it even offers advantages from the viewpoint of transaction management.

In the proposed concept, all components (and their processes, respectively) except the programming language adapter or the user interface are running on a centralized server. Additionally, each client has a local buffer (heap), where CoOMS objects can be stored temporarily.

An advantage of such an approach is that in the case of (direct) access to an object, it may already be in the buffer, and therefore no network traffic or involvement of the CoOMS server are necessary in order to retrieve the object. In fact, similar models (so-called workspace models) were proposed in order to isolate multiple transactions from each other. Here, modified objects are stored in the workspace, and only if the transaction is willing and permitted to commit, the modifications are flushed to the database. In fact, from the viewpoint of buffer management, this is the NoSteal option in the terminology of [Haerder 83]. If modifications are written to the database exactly at (but not after) commit, the Force option is followed additionally. Thus, we would not need undo and redo logging (but a two-phase commit [Bernstein 87]). From the viewpoint of transaction management, life seems rather easy in such an environment.

However, at least the NoSteal option appears to be inappropriate in the setting of CoOMS (or in systems supporting declarative, set-oriented access). Assume some modified objects stored in the heap, and let a user or application issue a (complex) query against an extension containing one of the modified objects. Assume a query where it is not obvious whether the modified objects will contribute to the result or not. Now, obviously, since the query is executed on the server, incorrect results can occur if the modified object is not taken into account, i.e. the application will not see the results of its own modifications. Therefore, the NoSteal option is not correct here, and thus the DBMS cannot be freed from undo logging (on the server).

However, redo information is not necessary in this case (the heap manager may follow the Force option). A further idea is to run closed nested transactions on the client only, but not on the server. In this case, the recovery and the log manager do not need to “understand” nested transactions, since rollback of subtransactions can be done locally on the client, and thus the server only has to abort top-level transactions. However, the price is implementation of (local) recovery and log managers and more space overhead (undo information has to be maintained by both, the server as well as the client). Recovery and logging for closed nested transactions, on the other hand, is essentially not much more complex than in the flat transaction case.

Additionally, the (centralized) lock manager still has to know about nested transactions.

Otherwise, i.e. if nested transactions are collapsed into a flat transaction, severe drawbacks result. Assume a leaf transaction accessing an object that is not already locked by any ancestor. Then, if the leaf transaction (or one of its ancestors) aborts later on, the lock may be released, thus letting other transaction access the object. If the operation occurs in what the lock manager views as a flat transaction, those locks cannot be released before commit or abort of the top-level transaction (at least not without further mechanisms). In another scenario with the same assumptions, let a deadlock occur. In this case, it might be sufficient to abort some subtransaction as a victim. However, if the lock manager knows flat transactions only, it has to abort entire top-level transactions. Thus, if lock management is not tailored to nested transactions, both, abort of more work than necessary as well as lower parallelism results.

Alternatively, one might include additional mechanisms (like checkpoints or non-strict, non-two-phase locking). However, this not only affects the simplicity of algorithms[4], but also implies higher implementation efforts. Therefore, one might reconsider whether the benefits of the approach actually outweigh its disadvantages.

6 Summary

Summarizing, CoOMS supports applications with closed nested transactions. Serializability is used as the correctness criterion for schedules of concurrent transactions. Internally, serializability is enforced by the use of strict two-phase locking, which takes object types as well as complex object hierarchies into account.

Multi-level transaction management is used in order to modularize the transaction subsystem in a clean way. For that matter, primitive transaction management is assumed to be provided on page level (by the responsible object server) besides locking and logging on the data model object level. Especially, we assume that synchronization and recovery are supported on the page level as well; as one possible technique the DB-Cache method may be used for recovery on the page level. However, other techniques may be chosen by the underlying object server (depending on the employed object server and on its functionality); what we actually assume is atomicity and durability of page level transactions; independent from the usage of the DB-Cache the multi-level approach will still work.

References

Abramowicz 89 — K. Abramowicz, R. Laengle, M. Ranft, T. Raupp, S. Rehm: *DAMOKLES Systemdokumentation*. FZI Publikation 13, Juli 1989.

Agrawal 83 — R. Agrawal, M.J. Carey, D.J. DeWitt: *Deadlock Detection is Cheap*. SIGMOD Record 13:2, 1983.

Astrahan 76 — M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, V. Watson: *System R: Relational Approach to Database Management*. ACM Transactions on Database Systems 1:2, 1976.

4. which is essential for implementing correct and reliable transaction management components

Aho 75 A.V. Aho, E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading MA, 1975.

Bayer 77 R. Bayer, M. Schkolnick: *Concurrency of Operations on B-Trees*. Acta Informatica 9, 1977.

Bernstein 87 P.A. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

Deux 91 O. Deux et al: *The O2 System*. In R.G.G. Cattell (Ed.): Special Issue on Next-Generation Database Systems, Communications of the ACM 34:10, 1991.

Dittrich 90 K.R. Dittrich, A. Geppert, V. Goebel, S. Scherrer: *The Data Definition Language of NO^2*. Project Report ITHACA.Unizh.90.X.4.#1, University of Zuerich, 1990.

Dumm 91 T. Dumm, T. Gorchs, M. Watzek: *The CoOMS Persistent Object Server. Requirement Specification, Functionality, and Architecture*. Project Report ITHACA.SNI.91.X.5#2, Siemens Nixdorf Informationssysteme, Berlin 1991.

Elhardt 84 K. Elhardt, R. Bayer: *A Database Cache for High Performance and Fast Restart in Database Systems*. ACM Transactions on Database Systems 9:4, 1984.

Garza 88 J.F. Garza, W. Kim: *Transaction Management in an Object-Oriented Database System*. Proc. ACM-SIGMOD Intl. Conference on Management of Data 1988.

Geppert 90a A. Geppert, K.R. Dittrich, V. Goebel: *Quod: A Query Language for NO^2*. Project Report ITHACA.Unizh.90.X.4#2, University of Zuerich, 1990.

Geppert 90b A. Geppert, K.R. Dittrich, V. Goebel, S. Scherrer: *An Algebra for the NO^2 Data Model*. Project Report ITHACA.Unizh.X.4#3, University of Zuerich, 1990.

Geppert 92 A. Geppert, K.R. Dittrich, S. Nittel, S. Scherrer: *Architectural Issues in Integrating CoOMS and OMS*. Project Report ITHACA.Unizh.X.5.#1.

Gray 78 J.N. Gray: *Notes on Database Operating Systems*. In R. Bayer, R.M. Graham, G. Seegmueller (Eds.): Operating Systems. An Advanced Course. Lecture Notes in Computer Science 60, Springer 1978.

Gray 81 J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, I. Traiger: *The Recovery Manager of the System R Database Manager.* ACM Computing Surveys 13:2, 1981.

Haerder 83 T. Haerder, A. Reuter: *Principles of Transaction-Oriented Database Recovery.* ACM Computing Surveys 15:4, 1983.

Haerder 87 T. Haerder, K. Rothermel: *Concepts for Transaction Recovery in Nested Transactions*. Proc. ACM-SIGMOD Intl. Conference on Management of Data 1987.

Haerder 87b T. Haerder: *Realisierung von operationalen Schnittstellen*. In P.C. Lockemann, J.W. Schmidt (Eds.): Datenbankhandbuch. Springer 1987.

Herrmann 89 U. Herrmann, P. Dadam, K. Küspert, G. Schlageter: *Sperren disjunkter, nicht-rekursiver Objekte mittels objekt- und anfragespezifischer Sperrgraphen*. Proc. BTW 1989.

Herrmann 91 U. Herrman: *Mehrbenutzerkontrolle in Nicht-Standard-Datenbanksystemen*. Informatik Fachberichte 265, Springer 1991.

Korth 83 H. Korth: *Locking Primitives in a Database System*. Journal of the ACM 30:1, 1983.

Mohan 90 C. Mohan, R. Obermarck: *Algorithms for the Management of Remote Backup Data Bases for Disaster Recovery*. IBM Technical Report, June 1990.

Mohan 92 C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz: *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. ACM Transactions on Database Systems 17:1, 1992.

Moss 81 J.E.B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD Dissertation, MIT, Cambridge MA., 1981. Reprint in [Moss 85].

Moss 85 J.E.B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

Moss 86 J.E.B. Moss: *An Introduction to Nested Transactions*. COINS Technical Report 86-41, Univ. of Mass. at Amherst, 1986.

Moss 86b J.E.B. Moss, N.D. Griffeth, M.H. Graham: *Abstraction in Concurrency Control and Recovery Management*. COINS Technical Report 86-20, University of Mass. at Amherst, 1986.

Lynch 83 N.A. Lynch: *Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control*. ACM Transactions on Database Systems 8:4, December 1983.

Reuter 87 A. Reuter: *Massnahmen zur Wahrung von Sicherheits- und Integritaetsbedingungen*. In P.C. Lockemann, J.W. Schmidt (Eds.): Datenbankhandbuch. Springer 1987.

Schwarz 84 P.M. Schwarz, A.Z. Spector: *Synchronizing Shared Abstract Data Types*. ACM Transaction on Computer Systems 2:3, 1984.

Weikum 84 G. Weikum, H.-J. Schek: *Architectural Issues of Transaction Management in Multi-Layered Systems*. Proc. Intl. Conference on Very Large Data Bases 1984.

Weikum 88 G. Weikum: *Transaktionen in Datenbanksystemen*. Addison Wesley, 1988.

Weikum 90 G. Weikum, C. Hasse, P. Muth: *Multi-Level Recovery*. ACM Symposium on Principles of Database Systems 1990.

Weikum 91 G. Weikum: *Principles and Realization Strategies of Multilevel Transaction Management*. ACM Transactions on Database Systems 16:1, 1991.