ARCHITECTURAL ISSUES OF TRANSACTION MANAGEMENT IN MULTI-LAYERED SYSTEMS

Gerhard Weikum , Hans-Jörg Schek

Computer Science Department
Technical University Darmstadt
D-6100 Darmstadt, West Germany

Abstract

The internal structure of current data base systems is ideally characterized by a hierarchy of multiple layers. Each layer offers certain specific objects and operations on its interface. Within this framework we investigate the transaction management aspects. It is shown that the System R kind of concurrency control can be generalized and an appropriate recovery method can be found by introducing a type of open nested transactions which are strongly tied to architectural layers. Especially, our approach includes application-specific levels on top of a data base kernel system. Up to now, most of the preprocessor solutions for so-called "non-standard" applications that have been proposed simply ignore aspects of concurrency control and recovery. We sketch different possibilities to realize transaction management in such a layered environment.

1. Introduction and Background

Two different directions of evolution in data base systems may be observed. On the one hand, the performance for commercial applications should be increased still further (/Ba83/). On the other hand, data base systems of the next generation are expected to support so-called "non-standard" applications such as CAD or office automation (/HR83b/, /SchP82/).

These two objectives, although different, have something in common. To achieve them it does not suffice to modify single system components locally, rather the overall architecture of data base systems is affected (/Kie83/, /LSch83/). An important part of the architectural considerations is concerned with transaction management (/Ba83/, /PrS83/).

Ideally we can look at the structure of modern data base systems as a hierarchy of "virtual machines". Each such "machine" is characterized by the objects and operations which are available at its interface. These are in turn implemented with the help of objects and operations of the layer one lower. Such a multi-layered architecture is described in /HR83a/. A variant of it forms the background of a data base kernel system which we plan to implement (/PSSW84/). One special design feature is that we use a single data model, the so-called NF^2-relational model, to describe conceptual as well as internal data structures in a uniform way (/SchS83/). In this model NF^2-tuples are the basis of complex structured objects which appear on both the storage structure and access path level and, in the context of "non-standard" applications, at the user interface.

The layered architecture of the projected data base kernel system is roughly described in the following figure 1.

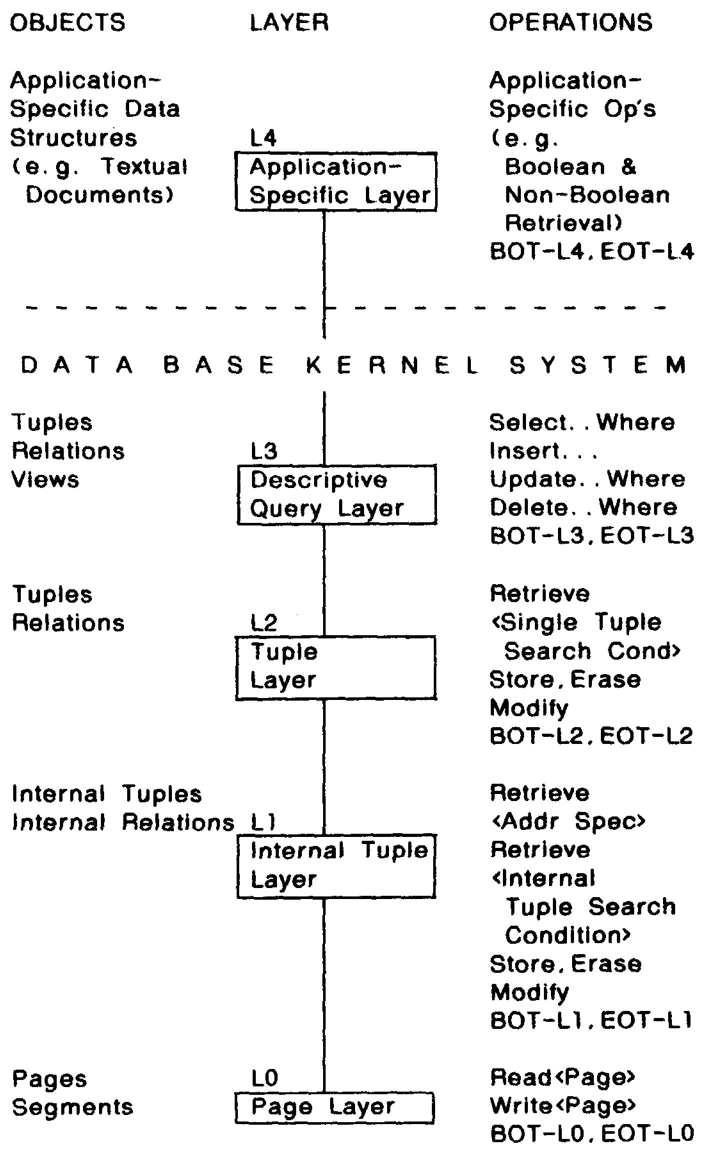


Figure 1: Multi-Layered DBMS
(BOT, EOT mean Begin of Transaction, End of Transaction resp.)

Apart from extensions inherent to the NF^2-relational model, the interface L3 in principle corresponds to the Relational Data System (RDS) of System R (/As76/): a single operation processes a set of (conceptual) tuples. L2 is comparable to the Research Storage System (RSS). Selection formulas are restricted to search conditions which can be evaluated locally on single tuples ("searchable arguments" in System R terminology) and update operations refer to current scan positions.
Whereas L2 operates on conceptual tuples as well, in L1 indexes are considered as internal (NF^2-) relations just as primary data are. L1 accordingly offers a special "address selection" on its interface. Such a layer has not been introduced in System R. Finally, underlying to L1 we have the segment and page structured storage module L0.

We suppose that not all of the requirements associated with "non-standard" applications can be performed by a single homogeneous system. It seems more realistic to extend a common kernel system by different preprocessors according to the application. Therefore, on top of L3 we will have an additional layer L4 offering application-dependent operations on data strutures such as documents, images, geometric shapes or matrices.

The question arises how to incorporate transaction management into such a multi-layered architecture. Conventionally, one would introduce a lock manager and a recovery manager into a particular layer. Very often the page layer L0 has been selected in available DBMS (e.g. UDS /Sie/ or SQL/DS /IBM/). The consequence for our architecture (fig. 1) would be that one user transaction (L4-level) is mapped into one L3-transaction which in turn corresponds to one L2-transaction and so on until we arrive at the one L0-transaction. This approach can also be found in System R: One RDS-transaction corresponds to one RSS-transaction, there is no different notion. A closer look into this subject recalls two important observations:

1. Although it seems so, transaction management is not a matter of a single layer. Even when located at L0 we depend on the fact that the next deeper layer (in this case the operating system) provides us with atomic operations (e.g. write a page).

2. A more careful inspection of the System R transaction management shows that we could understand the shadow page concept and the RSS-operation logging as a two-level recovery scheme (/Gr81a/, /Tr82/). Similarly we find that RSS tuple locking is complemented by page locking. Therefore, also the System R concurrency control mechanism is a two-level approach (/As76/, /Tr83/).

The idea of providing all levels of a multi-layered architecture with their own recovery mechanism has been proposed already in /Ve79/, but has not been pursued further as far as known. Also concurrency control aspects have not been discussed there. With this in mind we can state our

Problem: What are the fundamental possibilities for transaction management in a multi-layered data base system architecture? How can the System R approach be generalized to several or all layers of a system as sketched in fig. 1? Is it advantageous to construct a transaction at level (i+1) with more than one transaction at level i?

This paper tries to give first considerations on the above questions. In the next chapter we generalize the two-level approach for concurrency control to a multi-layered system. In chapter 3 we investigate related recovery aspects. Finally, in chapter 4 we show that these concepts can also be generalized to application-specific layers such as text retrieval on top of a data base kernel system.

2. Incorporation of Concurrency Control into Layered Architectures

2.1. Conventional Techniques

The most familiar and easy to understand concurrency control method is two-phase locking on page level (L0) with all locks held until End of Transaction (EOT) (cf. /HR83a/). Because of the rather coarse granularity the concurrency of transactions might be limited with this approach.

A respresentative of more sophisticated locking techniques is System R, which needs not necessarily treat pages as objects for concurrency control. Tuple locks are a possible option of System R. Therefore, in the following scenario of RSS-actions (see fig. 2), i.e. L2-operations according to fig. 1, no conflict occurs if t1 and t2 are two different tuples.

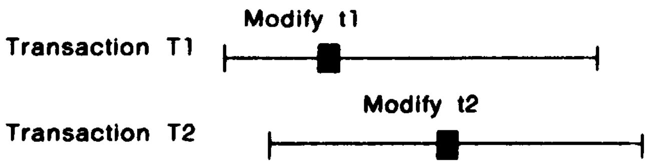


Figure 2: Operations at the L2-level (RSS-operation level)

As RSS-operations are nevertheless transformed into page accesses at runtime, the system still has to guarantee some sort of page level concurrency control. To demonstrate this necessity , let us assume that t1 and t2 are modified such that they afterwards require more space within the respective page (e.g. by increasing a variable length field). In the following we sketch a possible L0-execution of the above schedule. It might leave an inconsistent data base, if both tuples are stored in the same page p but the latter has free place only for one update.

Step 1: T1 reads t1 and checks the free place of p
Step 2: T2 reads t2 and checks the free place of p
Step 3: T1 modifies t1 within p
Step 4: T2 modifies t2 within p

System R avoids such situations and solves the problem by requesting page level locks for L0-operations and holding them until the end of the RSS-action (/As76/). This means that the system has a strict two-phase locking protocol for both, for layer L2 within the scope of each transaction and for layer L0 within the scope of each L2-operation (RSS-action). Regarding level L0, fig. 2 corresponds to the situation shown in fig. 3.

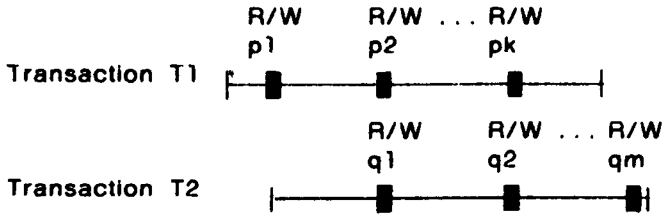


Figure 3: Operations at the L0-level (page level) (R/W means Read/Write of pages p1...)

Obviously, such a two-level locking mechanism excludes any concurrency anomalies (maybe, apart from phantoms). A formal proof of this statement is missing, however. Since page level locks are released at the end of each surrounding RSS-action, the method sketched above is called "open nested transaction" in /Tr83/ (compare also with /Gr81b/).

In spite of System R's rather fine locking granularity situations arise in which potential concurrency is prevented unnecessarily. This can be made clear looking at figure 3. With the mapping of tuple operations onto operations on data pages and possibly index pages, the concurrent execution of T1 and T2 has no delays only under the assumption that the referenced page sets are disjoint, i.e. if $\{p1, \dots, pk\} \cap \{q1, \dots, qm\} = \phi$. Otherwise one transaction must wait until a lock on a common page is released, which is at the end of the RSS-action that holds the lock. The first page which is accessed (exclusively) by both transactions causes a delay. In the worst case a conflict between T1 and T2 occurs at the first data page p1 (= q1). The compatibility of tuple locks would be meaningless in such a situation, because T2 would be blocked by T1 due to locking w.r.t. L0-operations. The probability for this type of conflict might be quite high, especially in case of many indexes to be maintained according to updates of primary data.

2.2. A Multi-Level Approach

There are two different ways to decrease the number resp. duration of delays occuring with the two-level approach sketched above. The first possibility is to apply special techniques for the synchronisation of operations on B^*-tree-like index structures. This direction generally means to utilize the knowledge that only a small number of well-known operations are allowed to operate on these special data structures. In the context of the previous discussion it means that we would have to distinguish between data pages and index pages.

The second possibility, which is our proposal, is an extension of the two-layer approach to more layers: the layer L1 could be made explicit and would get its own concurrency control. We assume that an L2-operation on one L2-tuple affects several (internal) L1-tuples. One L1-tuple, in turn, may affect several L0-pages. In the context of the previous discussion data tuples and index tuples would be regarded both as internal tuples to be managed by L1-operations. Notice that we would not differentiate between internal data tuples and index tuples to avoid unnecessary duplication of functions (/PSSW84/). In the following we discuss the various advantages of our multi-level approach.

The layer L1 would have to grant locks on internal tuples for the duration of L2-operations. Page level locks then could already be released at the end of each L1-operation. This situation is shown in fig. 4.

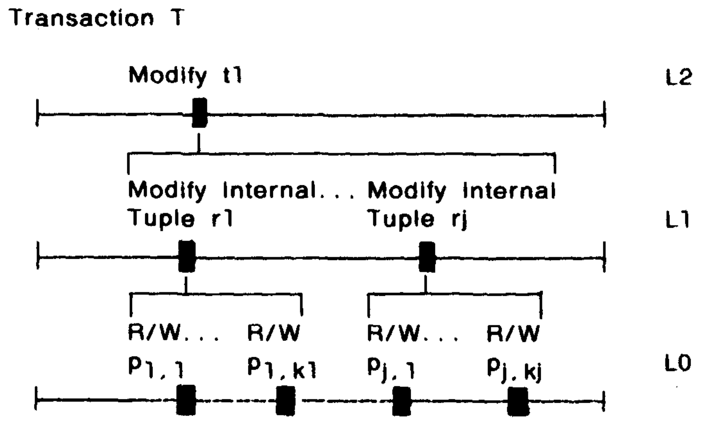


Figure 4: Open nested transactions for three layers (L2: Locks until EOT; L1: Locks for the duration of an L2-operation; L0: Locks for the duration of an L1-operation)

As we explained above, the worst case of the schedule described in figures 2 resp. 3 arises from an L0-conflict between T1 and T2. In this situation one of the two transactions would have to wait until the other's (con-)current L1-operation is finished, whereas without the additional locking at the L1-level the delay lasts for the duration of the surrounding L2-operation. Compared with System R the waiting period approximately decreases with the number of internal tuples to be modified per (conceptual) L2-tuple. The proposed extension of the RSS locking mechanism might pay off particularly for environments in which many indexes have to be maintained. As a critical point however we must consider the additional implementation overhead involved with a more sophisticated concurrency control method.

We could apparently generalize the concept of "open nested transactions" to any n-layered architecture. Each level L(i-1) regards the sequence of L(i-1)-operations corresponding to an Li-operation as a subtransaction of the comprising sequence of Li-operations. This subtransaction has to be serialized with other concurrent L(i-1)-subtransactions, that belong to different root transactions. The term "conflict of two operations" could be defined in a specific way for each layer so that we are not confined to two-phase locking nor to locking methods in general.

Another advantage of these "open nested transactions" is that so-called "semantic" concurrency control methods (e.g. /BGL83/, /Ga83/, /Ko83/, /Ly83/) might be applied. The notion of serializability is more than a pure syntactic criterion with such approaches. By considering (part of) the semantics of operations they lead beyond the simple concept of schedules as a sequence of "read's" and "write's". For example, one could have a special lock mode for each operation, whose compatibility is deduced from semantic properties (/Ko83/). As operations are layer-specific, the close relationship to the ideas presented in this chapter becomes clear.

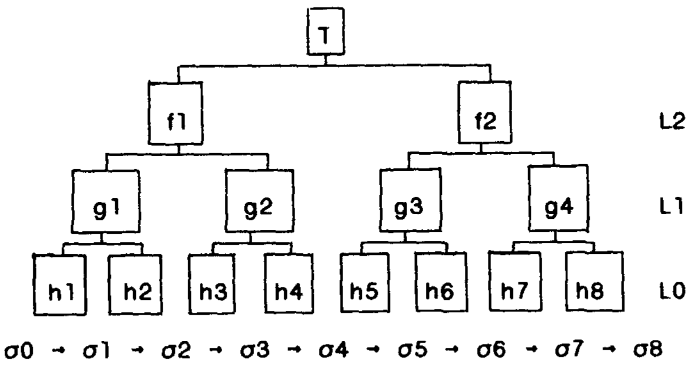
Finally we can apply this nested transaction concept also in an environment consisting of a data base kernel system and application-specific layers (cf. chapter 1). The mapping of application operations to the kernel interface, which is done by a "preprocessor", might again utilize the transaction management of the L3-layer to form subtransactions within a (longer) L4-transaction. We will return to this point in chapter 4.

3. Appropriate Approaches to Transaction Recovery

This section is devoted to the second branch of transaction management, that is recovery. Since it can not be regarded isolated from concurrency control, we must include aspects of the latter too.

3.1. A Transaction Model for Multi-Layered Architectures

A simple model of the execution of a transaction T in the multi-layered data base kernel system according to figure 1 looks like the following. (We restrict the discussion to the lower levels for the moment and will come back to the upper levels in chapter 4.)



T
f1
f2
L2
g1
g2
g3
g4
L1
h1
h2
h3
h4
h5
h6
h7
h8
L0
$\sigma 0 \rightarrow \sigma 1 \rightarrow \sigma 2 \rightarrow \sigma 3 \rightarrow \sigma 4 \rightarrow \sigma 5 \rightarrow \sigma 6 \rightarrow \sigma 7 \rightarrow \sigma 8$

Figure 5: An example transaction in its resolution through the layers ($\sigma 0$ ... are states of the volatile data base)

Each of the layers L2, L1 and L0 is represented by state-transforming functions fj, gj and hj respectively. For ease of explanation we assume a single-user mode for the moment. As usually we distinguish between a volatile data base and a permanent one, which both are modelled as elements σ_v resp. σ_p of a set Σ of states. State transitions primarily refer to the volatile data base. The only assumption about the propagation of updates to the permanent data base is that at the begin and at the end of each transaction we postulate $\sigma_v = \sigma_p$ to hold, which means that updates have to be propagated not later than EOT. So we can discard REDO measures from our discussion of soft crash recovery.

Let us assume that a system crash occurs in state $\sigma 7$ of the scenario of figure 5. At this time neither $\sigma_p = \sigma 0$ nor $\sigma_p = \sigma 7$ holds for the permanent data base σ_p in general. The objective of recovery is to reestablish state $\sigma 0$ through appropriate UNDO activities. We call a state $\sigma \in \Sigma$ "Li-consistent" if it is the outcome of a sequence of complete Li-operations starting from a state, e.g. $\sigma 0$, which, in turn, results from a sequence of complete previous transactions. Looking upon transactions as operations of an uppermost layer Ln (n = 3 in fig. 5 scenario) we could state as a rule: After a soft crash the data base system has to reestablish the last Ln-consistent state that has been reached (w.r.t. the volatile data base) before.

From the point of view of multi-layered architectures we might imagine that control is passed to a "recovery function" Ri: $\Sigma \rightarrow \Sigma$ for all layers in a bottom-up order (i = 0, 1, ..., n-1) (cf. /Ve79/). Each Ri is implemented solely with operations of the corresponding layer Li. The recovery mechanism works correct if with R0 starting from the surviving "after-crash-state" σ_p of the permanent database the equation

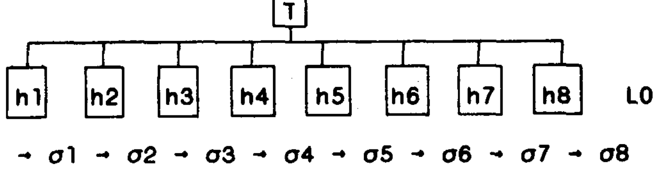
$$R_{n-1}(\ldots R_0(\sigma_p)\ldots) = \sigma 0$$

holds.

3.2. A Family of Recovery Algorithms

3.2.1. Classical Methods

In this section we consider recovery variants that could be understood in terms of our model. The architecturally most simple form of recovery is based on page logging. In such approaches R0 is responsible to restore the desired state $\sigma 0$, whereas all the higher level recovery functions Ri (i>0) are identity mappings on Σ. The relevant part of figure 5 looks as follows:



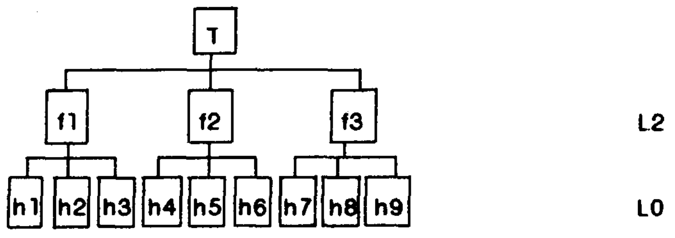
$\sigma 0 \rightarrow \sigma 1 \rightarrow \sigma 2 \rightarrow \sigma 3 \rightarrow \sigma 4 \rightarrow \sigma 5 \rightarrow \sigma 6 \rightarrow \sigma 7 \rightarrow \sigma 8$

Figure 6: Degeneration to page logging

The execution of page level operations hj is recorded, usually in form of their inverses hj^{-1} as so-called "before images", on a log file. Recovery then produces the state $R0(\sigma_p) = h1^{-1}(\ldots h7^{-1}(\sigma_p)\ldots) = \sigma 0$ after a crash in $\sigma 7$. A special property similar to "idempotency" (/Gr78/) or "restartability" (/Gr81b/) is required for L0-inverses: For every L0-operation hj and every state σ that does not "contain" hj the condition $hj^{-1}(\sigma) = \sigma$ must hold. The property is guaranteed for UNDO based on "before images" as well as for entry logging combined with so-called "log sequence numbers" (/Li79/).

Pure page logging is unsatisfactory for two reasons. First, rapid growth of the log file may cause serious problems especially with applications like CAD or office systems where objects are rather large and transactions are long. A primitive operation on a CAD object could trigger a multitude of pages to be modified (cf. /PrS83/, /KW83/). The second disadvantage of page level recovery is that it implies page level locking too. Otherwise updates of a successfully completed transaction could be lost due to another transaction aborting concurrently. The system would no longer guarantee isolated rollback. This well-known thesis about interrelations between recovery and concurrency control (/HR82/, /Tr82/) gives additional motivation to investigate transaction management in multi-layered architectures more thoroughly.

Both drawbacks we mentioned above are avoided in the System R recovery manager (/Gr81a/, /Tr82/) more or less. As the following figure shows, undoing transactions after a soft crash involves levels L2 and L0.



$\sigma 0 \rightarrow \sigma 1 \rightarrow \sigma 2 \rightarrow \sigma 3 \rightarrow \sigma 4 \rightarrow \sigma 5 \rightarrow \sigma 6 \rightarrow \sigma 7 \rightarrow \sigma 8 \rightarrow \sigma 9$

Figure 7: The System R recovery as a two-level approach (L2: RSS-operations; L0: page level operations)

The well-known shadow storage mechanism (/Lo77/) provides for atomicity of L2-operations because checkpoints (i.e. points in time to propagate updates) are always on RSS-action boundaries. Moreover, System R not just has the "all-or-nothing" paradigm for every single operation fj, but guarantees for any two L2-operations f1 and f2 that if the later one, f2, has been propagated to the permanent data base, then the updates of f1 must survive a system crash too. In /Wei84/ a more precise definition of this property is contained that can be applied to all levels of a layered architecture.

The recovery function R0 of the shadow storage thus ensures that after a crash in state $\sigma 7$ during the execution of transaction T (see figure 7) the L2-recovery R2 starts with one of the possible states $\sigma0$, $\sigma3$, or $\sigma6$, which all are L2-consistent. Since all RSS-calls fj are recorded on the log file, the R2 restart simply has to arrange the execution of all inverse operations fj^{-1} "contained" in the state $R0(\sigma_p)$ in reverse order. If, for example the most recent RSS-checkpoint was generated in state $\sigma3$ and the contents of the log file is <f1, f2>, the system must perform $f1^{-1}(\sigma3)$ in order to reestablish $\sigma0$. To prevent that $f2^{-1}$ is applied to $\sigma3$ either, the log file is organized with some sort of "log sequence numbers" (cf. /Gr81a/).

Invertibility of RSS-operations is the only prerequisite for this method to work. Notice that this requirement is nontrivial having the "DROP TABLE" statement in mind. It enforces the end of a System R transaction because the corresponding inverse operation cannot simply be constructed from a short log entry containing this RSS-call (/Tr82/).

3.2.2. Proposal for a Multi-Level Method

We have summarized the System R recovery mechanism because we will now be able to explain our generalization to n layers. The two-level System R recovery technique obviously is coordinated with the two-level concurrency control sketched in chapter 2. As we proposed that layers L2, L1, and L0 should contribute to concurrency control, it suggests itself to let participate also the L1-layer, but more generally all layers, in recovery too. The result of this idea is a system of nested transactions with levels naturally tied to architectural layers. Such a systematic approach can be extended to hierarchies of independent subsystems, for example for transaction management in application-specific preprocessors on a data base kernel system (see chapter 4).

On the lower end it is expected that future operating systems will offer some kind of transaction concept (/Tr83/, /BaS84/), which then could be utilized by data base systems to simplify their own transaction management. The interface L0 at the bottom of our data base kernel system of figure 1 offers atomic operations, but no real transactions. The difference is that the effect of successfully executed page operations might be lost afterwards if a system crash occurs and the data base buffer gets lost. To obtain transaction characteristics L0-operations additionally need durability, often called "persistency", and "logical indivisibility" w.r.t. concurrent actions. Whereas the latter is no problem, we must realize that persistency can only be achieved at the expense of a poor performance, either by forcing modified pages to disk immediately or by writing a REDO log record for each page update. A better solution is to have persistency for the comprising L1-operation merely, instead of L0. L1 could thus play the role of an "intelligent", (NF^2-) tuple oriented stable memory forming the basis of all higher layers. This stable memory could be implemented with satisfactory performance utilizing the "cache/safe" ideas of /Ba83/ and /El82/. In what follows we nevertheless assume a persistent L0- interface to simplify the discussion.

So we return to our main line that each layer L(n-1), ..., L0 helps to restore a transaction's initial state after a crash. If L0 guarantees persistency then the log file in the various intermediate states of figure 5 could look as follows:

State	Log File
$\sigma0$	< > i.e. empty
$\sigma1$	<h1>
$\sigma2$	<h1, h2> or <g1>
$\sigma3$	<g1, h3>
$\sigma4$	<g1, h3, h4> or <g1, g2> or <f1>
$\sigma5$	<f1, h5>
$\sigma6$	<f1, h5, h6> or <f1, g3>
$\sigma7$	<f1, g3, h7>

Figure 8: Log file entries in the example scenario

Due to L0-persistency the permanent data base is in state $\sigma7$ after a crash in $\sigma7$. Rollback of the considered transaction is done by applying the log's inverses, i.e. through $<f1, g3, h7>^{-1}$ = $<h7^{-1}, g3^{-1}, f1^{-1}>$. Execution of an inverse Li-operation belongs to the corresponding recovery function Ri. As the functions R(i-1), ..., R0 are transparent to the layer Li, we virtually achieve persistency for all layers including the transaction level Ln. The result of each Ri is the most recent L(i+1)-consistent state that was reached before the crash. For simplification figure 8 shows a global log file for all layers, but actually each level should record its operations autonomously instead.

The basic principle of each layer Li is to record all Li-operations of a surrounding L(i+1)-operation f until the latter is finished and L(i+1) has written a log entry for f. Then the Li-log is needed no longer and could be deleted. Since a crash might occur exactly at a time where both exist on the respective logs, f and its corresponding sequence of Li-operations, operations of all layers must be "careful", which means that, for example $<g1, h1, h2>^{-1}(\sigma2) = g1^{-1}(<h1, h2>^{-1}(\sigma2)) = g1^{-1}(\sigma0) = \sigma0$ holds.

As we already stated for the multi-layered concurrency control of chapter 2, an increased overhead must be expected with this kind of "hierarchical layer oriented logging". Therefore it is obvious to weaken the assumptions of our approach and let only selected layers contribute to recovery. This does not change the basic principle, as some levels are simply skipped when they do not contribute to recovery. Let Li and Lj (i>j) be two layers writing log records such that no intermediate layer Lm (i>m>j) participates in recovery. Lj has to record all level j operations belonging to the comprising Li-operation f currently executed. With the successful completion of f the Lj-log entries are deleted and their purpose is taken over by the entry "f" in the Li-log.

This approach is similar to the kind of nested transactions described in /Mo82/. The difference to /Mo82/ however is that in our concept nesting is strictly tied to architectural layers and not visible to the application programmer on the uppermost interface. Further, w.r.t. concurrency control we use the "open" type of nested transactions.

We finally describe the "selected layers logging" in an algorithmic form. Each layer Li has to understand an additional BOT- resp. EOT-call to indicate the begin resp. the end of the surrounding L(i+1)-operation. Thus, for each Li (i>0) and any Li-operation f we have the following generic actions:

```
action BOT:
  if $L_i$ is one of the selected recovery layers
    then prepare $L_i$-log
         (activation of a $L_i$-subtransaction)
    else call $L_{i-1}$ ("BOT")
  fi;

action f:
  if $L_i$ is one of the selected recovery layers
     and an $L_i$-subtransaction is activated
    then call $L_{i-1}$ ("BOT")
  fi;
  let <$g_1, \dots, g_k$> be the $L_{i-1}$-operation
      sequence implementing f;
  for j := 1 to k do call $L_{i-1}$("$g_j$") od;
  if $L_i$ is one of the selected recovery layers
     and an $L_i$-subtransaction is activated
    then write "f" to the $L_i$-log;
         call $L_{i-1}$ ("EOT")
  fi;

action EOT:
  if $L_i$ is one of the selected recovery layers
    then release $L_i$-log
    else call $L_{i-1}$ ("EOT")
  fi;

action $R_i$:
  call $R_{i-1}$;
  if $L_i$ is one of the selected recovery layers
    then let <$f_1, \dots, f_r$> be the
             contents of the $L_i$-log;
         for j := r to 1 do
           call $L_{i-1}$("$f_j^{-1}$");
           erase "$f_j$" from the $L_i$-log
         od
  fi;
```

For L0 the actions f are simply elementary operations, so that we have the following implementation:

```
action BOT:
  prepare $L_0$-log
    (activation of a $L_0$-subtransaction);

action f:
  write "f" to the $L_0$-log;
  execute f;

action EOT:
  release $L_0$-log;

action $R_0$:
  let <$f_1, \dots, f_r$> be the
      contents of the $L_0$-log;
  for j := r to 1 do
    execute $f_j^{-1}$;
    erase "$f_j$" from the $L_0$-log
  od;
```

It is remarkable that the recovery functions Ri call operations of the layer one below just as in normal processing mode of the data base system. In contrast to the regular mode no log records are written however to prevent undoing UNDO-operations in case of a second crash during restart. Ri depends also on the prerequisite of L0-persistency, since log records are deleted as soon as the corresponding inverse operation has been executed successfully.

The algorithm sketched above can be used for both undoing a single transaction due to deadlock or user abort as well as for crash recovery. Since each UNDO step of an Li transaction must reaquire certain system resources such as L(i-1) locks, special measures, i.e. like System R's "Golden" latch (/Gr81a/), should be taken to guarantee that every aborting transaction is eventually terminated. In the worst case an UNDO might result in a system crash when resources are exhausted and cannot be made available at runtime. However, this seems not to be uncommon practice.

4. Transaction Management in an IRS-Preprocessor to a (Kernel) DBMS

The objective of this chapter is twofold. First, it can be regarded as a concrete example and further justification for the open nested transaction approach described so far. Second, it analyses the aspects of transaction management of so-called preprocessor solutions to DBMS which are thought to support non-standard applications (/KL83/, /SRG83/, /Wo83/, /Sch84/). A general expectation seems to be that a preprocessor to an available DBMS does not need any concurrency control or recovery function since the underlying DBMS would be responsible for that.

In the following we will study this expectation more carefully. We will take the example of an Information Retrieval System (IRS) at layer L4 (fig. 1) as a preprocessor to a (kernel) DBMS (layer L3 in figure 1). We might also think of a DBMS like SQL/DS as a target which we map the IRS onto. (Actually an IRS preprocessor has been implemented on the System R prototype at the IBM Heidelberg Scientific Centre /EHPR81/ and experience was gained from that exercise.) For the following discussion we use IRS interface and L4-layer synonymously as well as DBMS interface and L3-layer.

4.1. Description of the IRS to DBMS Mapping

For the follwing discussion it is sufficient to regard only simplified objects and operations at the two layers of the IRS-DBMS-combination.

4.1.1. Objects and Operations at the IRS Interface

At the IRS interface we have the following main objects

```
document collection (DC)
document (D)                          (IRO)
term(T)
IR transaction (IRT)
```

A document collection is a set of documents. Each document consists of two fields. The first field is a document number (DNO), the second field is a set (TS) of terms. The DNO should identify a document. The operations which are important for the following are

```
INSERT (dc,doc)
DELETE (dc,doc)                       (IROP)
UPDATE (dc,doc,δtd,δti)
SEARCH (dc,doc,query,query-name,estimate)
NEXT (doc,query-name)
```

The parameter dc is name of a document collection and doc is a document of the type as described above. In case of an INSERT the DNO field will be defined after a successful insertion. In case of an UPDATE the DNO field contains the document number (previously found by a SEARCH or NEXT) to be changed. The update is defined by two sets of terms δtd and δti. The first contains the set of terms to be deleted, the second the ones to be inserted. In the DELETE case the DNO field of doc contains the document number to be deleted (also found by a previous search). In the SEARCH command the "query" parameter denotes a set Q of terms. A document is a match to Q if $Q \subseteq TS$ i.e. if all terms of Q appear in the set of terms TS of a document. The parameter "estimate" is a system estimated number of document matches to the given query and doc contains a first match.

One or more NEXT calls can be issued after a previous search to the same query (identified by "query-name") and the same documents type. It results in further matching documents, one for each NEXT ("Scan" through the matching documents).

An IRS transaction (IRT) is defined as a sequence of operations which is entered by an IRS user

```
IRT := BOT-L4; A1;A2;...;An; EOT-L4
```

Each operation Ai is one of those defined by (IROP) on objects defined through (IRO). Two options exists for EOT. One is "ABORT", the other is "COMMIT". The abort option is used when the changes made since the last BOT-L4 shall be undone. The commit option makes all changes since BOT-L4 visible to other users and persistent. This is the usual notion of a transaction.

We shall consider two examples of transactions later on:

```
IRT1 := BOT-L4;
        SEARCH(Q1);NEXT(...);...;
        SEARCH(Q2);NEXT(...)...;
        EOT-L4(COMMIT);
IRT2 := BOT-L4;
        SEARCH(...);UPDATE(...);
        INSERT(...);
        EOT-L4(COMMIT)
```

Obviously the first transaction is a typical IR search transaction. It has a first SEARCH with some query Q1, gets more matches with NEXTs and decides to modify the query, issue a next SEARCH with Q2 and a sequence of NEXTs for the new matches. The second is (in present systems) not a typical IR transaction. After some search the found document is updated, a new document is inserted and the transaction commits.

Note that the notion of a transaction seems to be important for new IR applications like the administration of office information: It may be necessary in our example for transaction IRT2 not to make visible a first update to a document unless a second document is successfully inserted. If the insert fails for some reason, also the first update must be undone.

4.1.2. Objects and Operations at the DBMS Interface

We regard only those objects and operations which will be necessary for the mapping of the previously described IRS objects and operations. We assume an interface with objects

```
Table or Relation (RL)
Tuple or Row (R)                      (DBO)
Attribute (A)
DBMS transaction (DBT)
```

As basic DBMS operations we introduce

```
DINSERT(rl,tup)
DUPDATE(rl,tup,new attribute values)      (DBOP)
DDELETE(rl,tup)
DSEARCH(rl,tup,query,query-id)
DNEXT(tup,query-id)
```

The meaning of operations and parameters is evident.

A DB transaction is again a sequence of actions DAi as they are defined in (DBOP) on the objects defined by (DBO).

DBT := BOT-L3; DA1; DA2; . . . ; DAk; EOT-L3

As usually this defines a unit of concurrency and recovery, now with respect to objects and operations at the DBMS interface.

4.1.3. Mapping of IRS Objects and Operations to DBMS Objects and Operations

In this section we describe the mapping of IRS operations on IRS objects to corresponding DBMS operations and objects. For that we let the IRS preprocessor impose the document interpretation on stored byte strings managed by the DBMS. For the DBMS we have variable length tuples (one document corresponds to one tuple) where the set of terms TS is represented as one (long) byte field BYTESTR1. The IRS preprocessor then knows how to interprete the byte string as a set-of-terms-structure. This procedure has been proposed recently as the approach of "abstract data types and abstract indices" /SRG83/ or as attribute level operations /Wo83/.

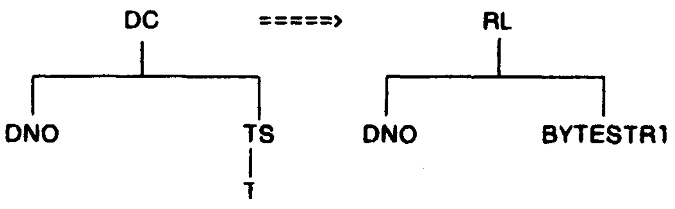


Figure 9: Mapping of an IRS object (layer L4) to a DBMS object (L3)

So far, we have considered only the mapping of IRS user objects. But in order to support IRS queries we must provide index support within the IRS preprocessor. We introduce the usual IRS term index IDC to a document collection consisting of an inversion on terms (figure 10).

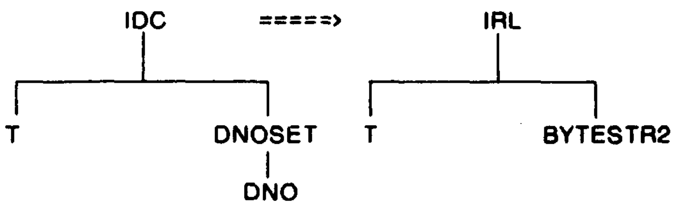


Figure 10: Mapping of an IRS term index to a DBMS object

Every index list consists of a term and a set (DNOSET) of all document numbers (DNO) pointing to a document which contains that term. Such an index list is mapped to a single tuple at the DBMS layer. This contains two fields: the first is the term, the second is BYTESTR2, a (long) bytestring as representation of the set of document numbers (DNOSET).

The IDC structure is not visible at the IRS interface but is inside the IR preprocessor and provides support for IR queries. It is also maintained within the IRS preprocessor in case of IRS updates, deletes or insertions.

We are now able to describe the transformation of IRS operations to DBMS operations. Assume that a document with a term set TS = {T1,T2,...,Tk} has to be inserted. Since we must maintain our term index this single user action has to be transformed in a sequence of DBMS actions as follows:

```
DINSERT(rl-of-dc, row-of-doc);
  /*DNO now definded*/
for all terms Ti ∈ {T1,...,Tk} do
  DSEARCH(irl-of-dc, indexrow, term='Ti');
  if found
  then
    Prepare new value (newbytestr) for BYTESTR2
        by inserting new DNO value;
    DUPDATE(irl-of-dc, indexrow, newbytestr)
  else
    Prepare new "indexrow";
    DINSERT(irl-of-dc, indexrow)
  fi
od;
```

This shows that one IRS (write) action produces (k+1) DBMS (write) actions. Similarily we will produce a sequence of DBMS actions for a single IRS query:

SEARCH(dc, doc, {T1',T2',...,Tq'})

is executed as

```
for all terms Ti' ∈ {T1',...,Tq'} do
  /*assume that all index pointer lists exist*/
  DSEARCH(irl-of-dc, indexrow, term='Ti');
  Determine a set CSET of DNO values
    which contain match candidates
od;
for "first" dnoj ∈ CSET do
  DSEARCH(rl-of-dc, row-of-doc, DNO=dnoj);
  Check whether delivered document
    (in row-of-doc) is a match;
  if not, try a "next" dnoj
od;
```

Again we see that a IRS query with q terms produces in the average a sequence of (q+1) DBMS searches to locate the first match.

These two examples of mappings between the IRS-layer and the DBMS-layer are sufficient to discuss now the question of transaction mapping.

4.2. Mapping of IRS Transactions to DBMS Transactions

4.2.1. One-to-one Mapping

The most simple way to map an IRS transaction onto the next layer is to generate exactly one DBMS transaction for it. This solution would indeed fully exploit the transaction management of the DBMS layer and no effort would be necessary with respect to concurrency control and recovery within the IRS preprocessor. The layer L4 would not be a recovery layer. Conflict tests would be performed on the basis of DBMS objects, not on the basis of IRS objects. This is now the same situation we had described in the previous chapters when we compared page level (L0) concurrency control and recovery with tuple oriented methods (levels L2 or L1). The disadvantages pointed out there are similar to the ones we have at this point:

First regarding concurrency control we see that transactions which are not conflicting w.r.t. IRS objects may be conflicting w.r.t. DBMS objects. This means that "pseudo conflicts" are generated by this transformation. In order to give an example for pseudo conflicts here we consider the two IRS transactions IRT1 and IRT2. We assume that IRT1 reads documents which are different from the ones which IRT2 changes or inserts. We denote the set of terms in the documents which IRT2 writes by TS1, TS2 (for the old and new term sets in the update) and TS3 (for the insert). Q1 and Q2 are the term sets which IRT1 uses in its query. Then, IRT1 und IRT2 are not in conflict if

$$Q_i \ \underline{not} \subseteq TS_j \quad (i=1,2;\ j=1,2,3)$$

Due to the transformation to the DBMS layer, however, they are already in (pseudo-) conflict if

$$(Q1 \cup Q2) \cap (TS1 \cup TS2 \cup TS3) \neq \emptyset$$

i.e. if there is a single common term. Simple probability calculations show that pseudo-conflicts occur with probabilities which are orders of magnitudes higher than for a real conflict. If locking is used in the DBMS also deadlocks may be frequent.

The second disadvantage is related to recovery. In the one-to-one transaction mapping recovery is at the layer L3 or below. Before-images or old values mean always the whole (long) bytestring BYTESTR2 which represents the set of document numbers in case of IR index tuples. On the other hand, if recovery would be done at layer L4 we could discard log entries for IR index maintenance completely (as in RSS /As76/).

As it can be seen now these arguments repeat similar arguments we had at the deeper layers already. Instead of the one-to-one transaction mapping we propose therefore a nested transaction approach.

4.2.2. Open Nested Transactions

We map an IRS transaction to a sequence of DBMS transactions by taking the sequence of DBMS actions which belong to a single IRS action as one unit. As we know already this approach needs concurrency control in layer L4 (i.e. in the preprocessor). A method which can be applied here is the predicate oriented locking approach using signatures /DPS83/. The details are not relevant for our discussion here but the important fact is that such an IRS concurrency control produces a sequence of operations which is equivalent to some serial execution of the transactions. The next layer, now, is responsible for the correct execution of these operations which in turn at that layer are again sequences of DBMS operations. But the proper execution of these is guaranteed by the DBMS transaction management function.

We also know from the previous discussion that recovery is desirable in the IRS-preprocessor. For that we need the inverse Ai^{-1} of each IRS operation Ai which changes IRS objects. E.g. the inverse of a INSERT is a DELETE. According to the well-known rule that the undo information - in our case Ai^{-1} to a write operation Ai - must be saved before the objects related to Ai are (over-) written we must introduce an additional log data set to a document data base. In order to be sure that log records are safely written before we commit any changes on our objects we can again utilize the DBMS transaction management: A (IRS) log record (i.e. for the IRS transaction abort facility) is appended to the sequence of DBMS operations of any IRS write operation. According to the algorithm of chapter 3 the transformation of an IRS transaction onto the DBMS layer now looks like (fig. 11):

$$\begin{bmatrix} \text{BOT-L4} \\ \\ \cdot \\ \cdot \\ \cdot \\ \text{Ai} \\ \\ \cdot \\ \cdot \\ \cdot \\ \text{EOT-L4} \\ \\ \end{bmatrix} \quad \texttt{===>} \quad \begin{bmatrix} \text{BOT-L3,} \\ \text{DINSERT(Log-L4, "BOT-L4"),} \\ \text{EOT-L3;} \\ \cdot \\ \cdot \\ \cdot \\ \text{BOT-L3, DAi1, DAi2, DAik,} \\ \text{DINSERT(Log-L4, Ai}^{-1}\text{), EOT-L3;} \\ \cdot \\ \cdot \\ \cdot \\ \text{BOT-L3,} \\ \text{DINSERT(Log-L4, "EOT-L4"),} \\ \text{EOT-L3;} \end{bmatrix}$$

Figure 11: Mapping of one IRS transaction into a sequence of DBMS transactions

Notice that in addition to log tuple writes for the inverse operations we must write a log tuple for the begin and for the end of an IR write transaction.

Let us again look into the previous IRT2 example. We assume that there is a system crash within the DBMS transaction belonging to the IRS operation INSERT. After restart the DBMS recovery component would produce a consistent DBMS state: the changes of the transaction belonging to the UPDATE would be - if necessary - redone and eventual changes caused by the transaction to the INSERT would be undone by the DBMS recovery component. Next, the IRS recovery component would find no "EOT-L4" log tuple of IRT2 and therefore, would undo the changes of the update by reading the related log tuple and perform the inverse operation which was stored in this log tuple. After this the data are in a consistent IRS state.

This example shows that the cost for increased parallelism and simpler logging may be more effort in case of recovery after a crash or for transaction abort. This may also be the price for a clean concept of transaction management through layers in general.

5. Conclusion

In this paper we investigated architectural issues of transaction management in layered systems. In particular, we generalized the System R two-level kind of "open nested transactions".

Our primary objective w.r.t. multi-user control was to increase concurrency by avoiding "pseudo-conflicts" that could occur, especially in an application-specific preprocessor on top of a data base kernel system, due to real conflicts in lower levels. Though there is much research on algorithms with this purpose, there is still no theoretical framework of concurrency control in the case of layered architectures. The work of e.g. /BGL81/ or /Ly83/ may be steps into this direction.

Nested transactions were also considered useful to structure recovery. Starting from hardware characteristics every additional layer Li increases the degree of resistancy with the aid of certain atomic L(i-1)-operations, finally achieving the classical transaction concept for the uppermost level. This possible generalization of the System R approach has up to now not been examined, though it might turn out as a key concept for incorporation of transaction management into the architecture of data base systems. An attempt to categorize recovery in layered systems, based on a more refined version of the transaction model described in chapter 3.1, is part of /Wei84/.

Further research efforts are necessary to develop a theoretical basis and to investigate practical applications of transaction management in layered systems.

Acknowledgement

We would like to thank Liz Klinger for the excellent preparation of this paper.

References

/As76/
M.M.Astrahan et al., System R: Relational Approach to Database Management, TODS Vol.1 No.2, 1976
/Ba83/
R.Bayer, Database System Design for High Performance, Proc. of the IFIP Conference, Paris 1983
/BaS84/
R.Bayer, P.Schlichtiger, Data Management Support for Database Management, Acta Informatica Vol.21 No.1, 1984
/BGL81/
P.A.Bernstein, N.Goodman, M.-Y. Lai, Laying Phantoms to Rest, Proc. IEEE COMPSAC Conference 1981
/BGL83/
P.A.Bernstein, N.Goodman, M.-Y.Lai, Analyzing Concurrency Control Algorithms When User and Systems Operations Differ, IEEE Transactions on Software Engineering Vol. SE-9 No.3, 1983
/DPS83/
P.Dadam, P.Pistor, H.-J.Schek, A Predicate Oriented Locking Approach for Integrated Information Systems, Proceedings of the IFIP Conference, Paris 1983
/EHPR81/
R.Erbe, F.Höckenrainer, R.Poloczek, B.Ruhbach, SQL-TR: A System R Extension for Text Retrieval, Internal Report, IBM Heidelberg Scientific Centre, 1981
/El82/
K.Elhardt, The Data Base Cache: Design Principles, Algorithms, Characteristics (in German), Doctoral Thesis, available as: Technical Report TUM-I8208, Technical University Munich, 1982
/Ga83/
H.Garcia-Molina, Using Semantic Knowledge for Transaction Processing in a Distributed Database, TODS Vol.8 No.2, 1983
/Gr78/
J.Gray, Notes on Data Base Operating Systems, in: Operating Systems - An Advanced Course, LNCS 60, Springer-Verlag 1978
/Gr81a/
J.Gray et al., The Recovery Manager of the System R Database Manager, ACM Computing Surveys Vol.13 No.2, 1981
/Gr81b/
J.Gray, The Transaction Concept: Virtues and Limitations, Proc. of the VLDB Conference, Cannes 1981
/HR82/
T.Härder, A.Reuter, Principles of Transaction Oriented Database Recovery - A Taxonomy, Technical Report 50/82, University Kaiserslautern 1982
/HR83a/
T. Härder, A. Reuter, Concepts for Implementing a Centralized Database Management System, Proc. International Computing Symposium, Nürnberg 1983
/HR83b/
T.Härder, A.Reuter, Database Systems for Non-Standard Applications, Proc. Internal Computing Symposium, Nürnberg 1983
/IBM/
SQL/Data System, Concepts and Facilities, IBM Corporation, Form No. GH 24-5013, 1981
/KW83/
R.H.Katz, S.Weiss, Transaction Management for Design Databases, Technical Report #496,

Computer Science Dept., University of Wisconsin, 1983
/Kie83/
W. Kießling, Data Base Systems for Computers with Intelligent Subsystems: Architecture, Algorithms, Optimization (in German), Doctoral Thesis, available as: Technical Report TUM-I8307, Technical University Munich, 1983
/KL83/
W. Kim, R. Lorie, Nested Transactions for Engineering Design Databases, Research Report RJ 3934, IBM San Jose, 1983
/Ko83/
H. F. Korth, Locking Primitives in a Database System, Journal of the ACM Vol. 30 No. 1, 1983
/Li79/
B. G. Lindsay et al., Notes on Distributed Databases, Research Report RJ 2571, IBM San Jose, 1979
/Lo77/
R. Lorie, Physical Integrity in a Large Segmented Database, TODS Vol. 2 No. 1, 1977
/LSch83/
V. Lum, H.-J. Schek (Chairmen), Complex Data Objects: Text, Voice, Images: Can DBMS Manage Them ?, Panel Discussion, Proc. of the VLDB Conference, Florence 1983
/Ly83/
N. A. Lynch, Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control, TODS Vol. 8 No. 4, 1983
/Mo82/
J. Moss, Nested Transactions and Reliable Distributed Computing, Proc. 2nd IEEE Symposium on Reliability of Distributed Software and Database Systems, 1982
/PSSW84/
H.-B. Paul, H.-J. Schek, M. Scholl, G. Weikum, Considerations on the Architecture of a "Non-Standard" Data Base Kernel System (in German), Internal Manuscript, Technical University Darmstadt, 1984
/PrS83/
U. Prädel, G. Schlageter, Concurrency Control in Integrated Information Systems: A Survey of Problems, Technical Report, University of Hagen, 1983
/SchP82/
H.-J. Schek, P. Pistor, Data Structures for an Integrated Data Base Management and Information Retrieval System, Proc. of the VLDB Conference, Mexico 1982
/SchS83/
H.-J. Schek, M. Scholl, The NFü-Relational Algebra for Uniform Manipulation of External, Conceptual and Internal Data Structures (in German), in: J. W. Schmidt (ed.), Sprachen für Datenbanken, IFB 72, Springer-Verlag 1983
/Sch84/
H.-J. Schek, Nested Transactions in a Combined IRS-DBMS Architecture, to appear in: Proc. 3rd BCS/ACM Symp. on Research and Development in Information Retrieval, Cambridge 1984
/Sie/
UDS Version 3.2, Reference Manual Package, Siemens AG, Munich 1982
/SRG83/
M. Stonebraker, B. Rubinstein, A. Guttman, Application of Abstract Data Types and Abstract Indices to CAD Data Bases, Proc. "Engineering Design Application", Database Week, San Jose 1983
/Tr82/
I. L. Traiger, Virtual Memory Management for Database Systems, ACM Operating Systems Review Vol. 16 No. 4, 1982
/Tr83/
I. L. Traiger, Trends in Systems Aspects of Database Management, Proc. 2nd Int. Conf. on Databases (ICOD-2), Cambridge 1983
/Ve79/
J. S. M. Verhofstad, Recovery Based on Types, in: G. Bracchi/G. M. Nijssen (eds.), Data Base Architecture, North-Holland Publ. 1979
/Wei84/
G. Weikum, Transaction Recovery in Data Base Systems with Layered Architecture: New Approaches to a Categorization (in German), Internal Manuscript, Technical University Darmstadt, 1984
/Wo83/
E. Wong, Semantic Enhancement through Extended Relational Views, Proc. 2nd Int. Conf. on Databases (ICOD-2), Cambridge 1983