appeared in: Proc. 23rd Annual Hawaii International Conference on System Sciences HICSS-23, Kona, Hawaii, Vol. II, pp. 472-480

Preserving Consistency in Nested Transactions

H. Schöning

University Kaiserslautern, Department of Computer Science,
P.O. Box 3049 , D-6750 Kaiserslautern, West Germany

Abstract

The support of advanced applications such as CAD/CAM by so-called non-standard database systems (NDBS) has emerged as an important direction in database system research. Since in these areas complex integrity constraints play an important role, NDBS have to provide a flexible mechanism in order to deal with them. For this purpose, the nested transaction concept developed for advanced applications is combined with a concept of *levels of consistency* . Each level of a nested transaction hierarchy is associated with a specific type of consistency, which is guaranteed at this level.

The implementation is based on the existence of a recording routing (called when consistency violations are found) and a checking routine (called at the end of a (sub-) transaction) for each type of consistency. The approach allows for tailoring the actions of both routines to the specific needs imposed by a particular type of consistency. The overall flexibility and the extensibility of the approach are illustrated by its implementation in the PRIMA project.

1. Introduction

The support of advanced applications such as CAD/CAM, VLSI, etc. by so-called non-standard database systems has emerged as a main direction in database system research. Since in these areas complex consistency constraints play an important role, database management systems (DBMS) developed for these applications have to provide a flexible mechanism in order to deal with them. Particularly, powerful means for the specification of consistency constraints and an efficient and extensible consistency preservation checking mechanism have to be implemented. Furthermore, in most of these application areas, the user must be able to react in accordance to consistency violations which eventuate: In a design environment, it is unacceptable to roll back a transaction lasting several weeks, if the consistency check fails. Instead, the system has to provide means to correct the inconsistencies in dialogue with the designer.

Independent of the requirements for consistency checking, elaborated nested transaction concepts have been developed [1, 2] tailored to advanced applications. In this paper, we try to connect the two concepts of nested transactions and consistency maintenance. In a conventional DBMS (this term will be used throughout the paper to describe DBMS used for traditional applications, e.g. banking applications) a “classical” (flat) transaction groups together a set of operations which are processed sequentially, while guaranteeing the following (“ACID”) properties [3]:

- *A*tomicity : Either all operations are executed or none of them.
- *C*onsistency : If a database is consistent before a transaction is executed, it is again consistent afterwards.
- *I*solation : If some transactions run concurrently, there is a sequential schedule of these transactions that yields the same result.
- *D*urability : After a transaction has finished, its effects are persistent.

Note that [3] distinguishes between *isolation* and *consistency* in contrast to other papers (e.g. [4, 5, 6]). The aspect of consistency as mentioned in [3] is often called semantic integrity, but we will differentiate between the two notions *consistency* and *integrity* in the following. The next chapter shows our definition of the two notions. Chapter 3 recalls some classifications of consistency. Our proposal for the combination of the concepts of consistency and nested transactions is presented in chapter 4. Finally, we discuss the design of a consistency manager and give an example for an implementation.

2. Consistency and Integrity

We now specify how the notions “consistency” and “integrity” are used in the following. A database represents a part of the real world. For this purpose, a function M_S for mapping real world states to database states is (at least conceptually) defined. It is reflected by the database schema, which describes the data objects to be dealt with as well as the associated data formats. In order to model the real world more exactly, schema specifications may allow for the definition of attribute domains or even consistency constraints. Thus, the DBMS can automatically guarantee these constraints. A database that corresponds to the description in the according schema and which obeys all constraints is called *(schema) consistent* . The database itself is called a correct *instantiation* of the schema. The word *integrity* on the other hand is used to express the correspondence of database state and real world state. Hence, consistency can be checked by the DBMS, while integrity cannot. The following example will illustrate our use of both notions in more detail.

Imagine a banking application, where a client draws 100$ from his account in the real world. The clerk, however, types 10$ instead of 100$ when entering the change into his database. The new database state is consistent, because it is a correct instantiation of the database schema. It even corresponds to a *possible* real world state (which is mandatory, if the schema is correct and complete), but the integrity of the database is violated in that it does not reflect the real world any longer.

Figure 1 shows this in a more abstract way: S_1 corresponds to the original state of the account. It is changed by function C in the real world giving S_2 (the account shows 100$ less). S_1 is represented by the database state DS_1, and if the clerk types the right amount (i.e. uses the correct mapping function from C to DC_1) the new database state is DS_2, which is the correct representation of S_2. If the clerk, however, types 10$ instead of 100$, he uses another mapping function (DC_2), resulting in the new database state DS_3, which is the correct representation of a possible real world state S_3, and hence is a consistent database state. Integrity, however, is violated, since the database does not reflect the real world state any longer.

In the literature, the two notions consistency and integrity are used in very different ways. While many authors (e.g. [7]) use only one of these notions, [8] distinguishes both notions by the complexity of represented correlations. [9] uses *consistency* for the fact that redundant data do not contradict one another, while *integrity* characterizes the correctness of the data.

In the following, we will only address consistency in the sense defined above. For this purpose, we will discuss some classifications of consistency constraints made for conventional DBMS and the according implementations. Afterwards, we show how the various types of consistency can be ordered in a hierarchy corresponding to that established by nested transaction concepts in DBMS for advanced applications.

3. Consistency in Conventional DBMS

The problem of consistency preservation has been investigated in conventional DBMS for a long time. [10] for instance proposed a so-called *integrity subsystem* for System R, which was based on the trigger concept. Besides various implementation descriptions, there are some more fundamental papers classifying consistency constraints according to several criteria (cf. [11]). The most relevant criteria in the context of this paper are:

- Constraints can **restrict transitions**, e.g. prohibit the transition from married to unmarried, or **states**, e.g. restrict the possible values of a field.
- The **scop** e of a constraint may be a **single field** (which is restricted to a certain range), a **tuple** (expressing dependencies among the fields within a tuple), a **relation** (describing a certain range for the value of an aggregation function applied to the relation), etc.
- A constraint may be **local** (i.e., it has to be fulfilled within the transaction under consideration) or **global** (i.e., it has to be fulfilled when all work is finished, where the notion “all work” is usually not covered by a corresponding mechanism in the DBMS).
- A constraint may be **immediately** checked after an update operation (e.g. range checks can be performed after each update) or its check may be **deferred** (e.g. to the end of the corresponding transaction). An example of the latter is referential integrity in a relational DBMS, which may be violated by one operation, but cured by the following one within the same transaction.

There are some proposals to implement the maintenance of consistency in a DBMS. The most prominent of them is the trigger concept (e.g. [12, 13]). Actions which check for consistency violation are triggered by changes to the database. If the end of a transaction also triggers an action, consistency preserving transactions can be implemented. [13] criticizes the mere trigger concept and proposes an extended event/trigger mechanism that coexists with transactions. The coupling of transactions to this mechanism, however, is not inherent in the proposal of [13].

4. Consistency and Nested Transactions

Advanced applications of DBMS, such as CAD/CAM or VLSI design require extensions of the “conventional” transaction concept. For these applications, nested transactions have been introduced for various reasons [1, 2], such as concurrency within a transaction, modularity, and failure isolation. A nested transaction consists of a tree of transactions. The root of this tree is called *top-level transaction*, and has the ACID properties. All other transactions in this tree are so-called sub-transactions, which have the properties *isolation* and *atomicity*. Due to isolation, a transaction may invoke several sub-transactions concurrentl*y*. If a sub-transaction commits, it passes all lock acquired during its lifetime to its parent transaction rather than releasing them. *Durability* is not guaranteed for sub-transactions, since if a parent transaction fails, the sub-transactions are rolled back even if they have already finished. A natural application of this concept associates a transaction level to each layer of a DBMS, as indicated in figure 2: Each operation at one layer is a transaction at a lower system layer. We now discuss the property *consistency.*

Since we can identify several levels of operations corresponding to the system’s layers (a SELECT operation is at a higher level than an ENTER_IN_B_TREE operation), we can also introduce several

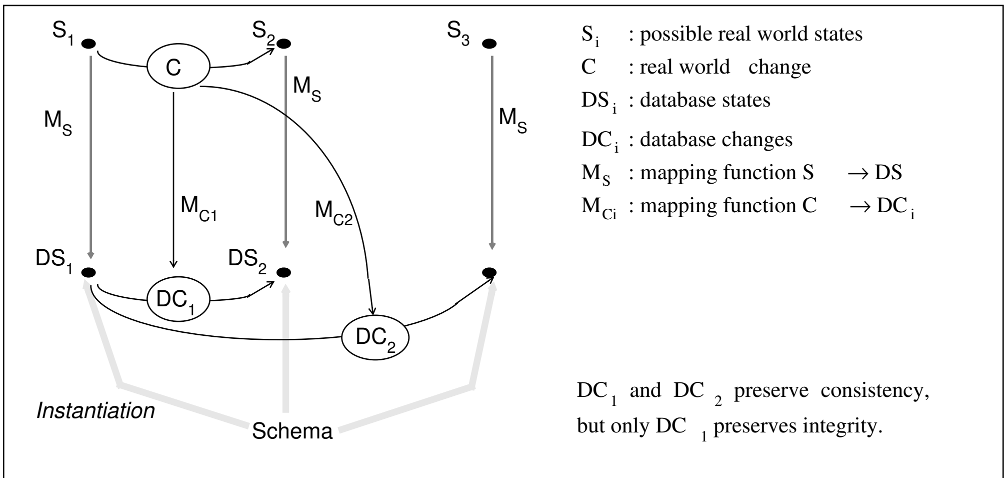

Figure 1: The difference between integrity and consistency

levels of consistency, one for each level of operations. [7] introduces at least two levels of consistency with the introduction of "local" and "global integrity" which are maintained by "integrity transactions".

The existence of different levels of consistency can be illustrated by the use of redundant data: In a banking application, a record containing the sum of all accounts is usually updated whenever an account is modified. This redundant information obviously obeys the consistency constraint that at the end of each transaction the value of this record must be equal to the sum of all accounts. Nevertheless, a single operation in a relational DBMS cannot preserve this consistency. At a lower level, B-tree entries are redundant data, too. The consistency of B-tree entries and data *(storage structure consistency)*, however, is guaranteed by each relational operation. Thus, we have identified two levels of consistency concerning redundant data.

In general, we can identify multiple levels of consistency, and corresponding operations (or operation sequences) that guarantee the corresponding type of consistency. It is worth noticing that consistency preservation at the lower levels of a DBMS is commonly embedded in the DBMS code, while it is separately handled at the "transaction layer" and the layers above. Some systems simply neglect the automatic consistency control, while others offer mechanisms almost independent of the transaction management, as mentioned above.

The aspects of consistency are excluded from considerations in many proposals for a nested transaction concept. Our idea is to combine the concept of nested transactions with the concept of the different consistency levels in order to get an integrated transaction management system for advanced applications. The basic assumption is that each sub-transaction potentially guarantees a certain type of consistency. Figure 2 illustrates the correspondence of architectural layers, transaction hierarchies, and consistency levels for the NDBS PRIMA which is described later in the text: Three architectural layers are shown, each involving sub-transactions in a transaction hierarchy. Associated with these transaction levels there are types of consistency guaranteed by the according transactions. We will explain the operations shown in figure 2 later.

The concept of nested consistency levels has some impact on the classification of consistency constraints mentioned above:

The differentiation between "global" and "local" consistency constraints must now be seen in relation to the levels: what is global to one level is local to a higher level (e.g. in Figure 2 cardinality restrictions are global consistency constraints for the data system level, but are local to the ADT level).

Analogously, a consistency constraint which is "deferred" with respect to a certain level is "immediate" at a higher level. For example, the referential integrity is a deferred consistency constraint for access system operations, while it is immediate for data system operations.

In the next chapter, we will show how our investigations concerning several levels of consistency lead to the design of a separate system component responsible for consistency preservation.

5. Design of a Consistency Manager

Traditionally, there is a component responsible for the isolation aspect of a transaction (a lock manager, for instance). The same holds for the aspects persistence and atomicity (logging and recovery component). In contrast to this, a component responsible for the consistency is hardly ever integrated. If consistency is regarded at all, the corresponding code is embedded in the overall DBMS code or there is no connection to the transaction management (e.g. [14, 15]). To overcome this imbalance in the handling of the four aspects of a transaction, we introduce the component "consistency manager" which has the task to control consistency preservation within a transaction. Its task is to guarantee a consistent database state at the end of a transaction with respect to the corresponding level of consistency. The design of this component has to be able to answer the following questions:

- How can the consistency manager test for consistency violations?

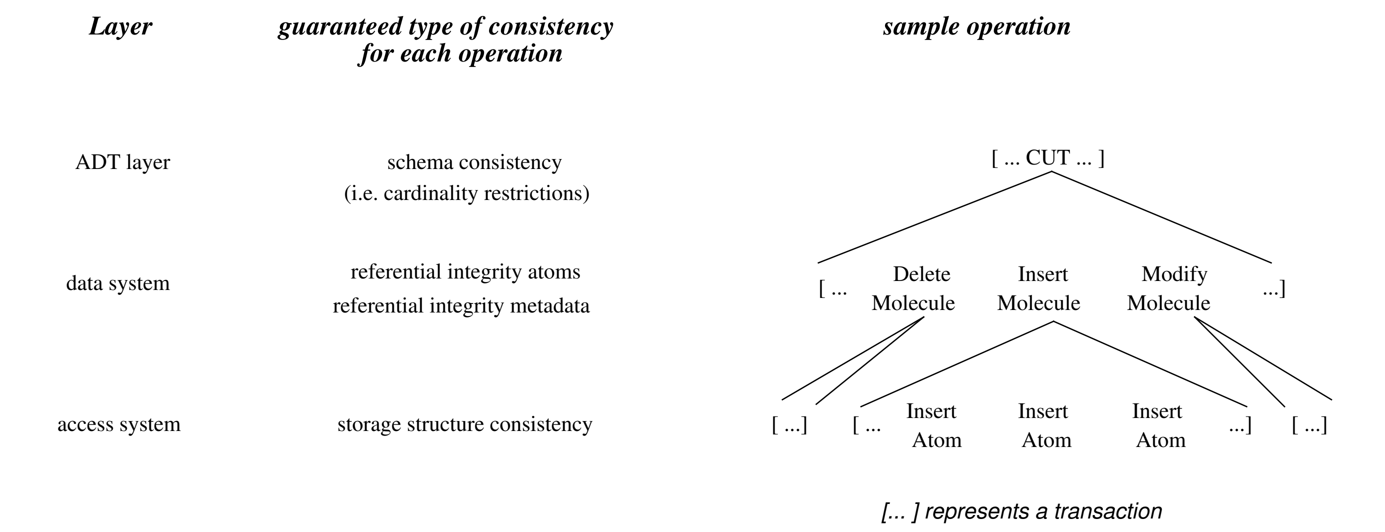

Figure 2 : Architecture, transaction hierarchy and consistency guarantee in PRIMA

- How should the consistency manager deal with consistency violations, and
- which types of consistency can be checked by the consistency manager?

Testing for Consistency Violations

Checking consistency can be quite complex if the whole database has to be considered. In many cases, the operations executed within a transaction give strong hints as to where consistency might be violated. For example, the consistency of B-tree entries need not be checked, if the corresponding relation has not been modified at all. These hints have to be used in order to allow efficient processing of consistency checking. To enable this, each update operation calls the consistency manager to convey its modifications, in analogy to the way log information is written or locks are requested. The consistency manager extracts the information relevant for consistency checking and preserves it until checking time.

The information collected by the consistency manager must be maintained according to the nested transaction semantics. I.e. the information collected within one transaction must be deleted in the case of an abort of this transaction. At the end of a transaction, the collected information for the types of consistency guaranteed by this transaction level is used to check for consistency. The other information must be passed on to the parent transaction. This corresponds to the way, locking information is handled in nested transactions.

Checking can already be initiated at the time consistency violation information is collected, thus making the check easier and less time-consuming. This can be done in the following ways:

- Consistency violations which mutually neutralize one another and belong to the same transaction can be forgotten during collection.
- In the case of self-correcting consistency constraints (see below), the correction can be initiated when inconsistencies appear. Then, the checking at the end of transaction only has to control whether all corrections have been successful.

Actions in the Case of Consistency Violations

If a particular type of consistency is violated at the end of the corresponding guaranteeing transaction, several reactions are possible:

- The consistency manager forces the transaction to abort. The database is thus rolled back to a consistent state. This straightforward solution, however, is unacceptable for transactions that last longer than a few seconds, particularly for design transactions which can last days and weeks.
- The consistency manager can return an error code to the transaction manager which is performing the call (“passive behavior”). While a commit request traditionally has only two possible results (success or abort), we allow a third reaction in this case: the transaction manager rejects the commit and delivers information about consistency violations. Thus, the component which wants to finish the transaction can decide how to proceed: either to correct the inconsistent state or to explicitly abort the transaction.
- In some cases, the consistency manager can automatically enforce a consistent state of the database (“active behavior”). Then, the corresponding transaction is aborted only if this automatic correction fails. This reaction corresponds to the self-correcting consistency constraints mentioned above.

Types of Consistency that can be Checked by the Consistency Manager

For each type of consistency, the way in which to collect information and how to deal with consistency violations may differ. Furthermore, it does not seem to be appropriate to restrict the types of consistency that are to be controlled. To allow for an individual strategy for each type of consistency constraint, we do not embed the checking and information collecting in the consistency manager code, but introduce two “independent” routines for each of them:

- The *recording routine* is responsible for the collection of consistency violation information. It receives the data touched by the operation under consideration, extracts consistency violation informations and collects them (possibly in one of the “intelligent” ways listed above). This routine is invoked by the consistency manager every time an operation at the corresponding or a lower level jeopardizes the according type of consistency.
- The *checking routine* performs the consistency checks at the end of a transaction of the corresponding level. If the consistency constraint is self-correcting, the checking routine corrects the consistency violations. Otherwise, if the consistency check fails, the transaction manager is informed to refuse the corresponding commit request.

To associate the routines to the various levels of consistency, there is a consistency mapping table interpreted by the consistency manager. Since this table can easily be manipulated, our approach is highly extensible. New types of consistency can easily be added to the table and be made mandatory to a certain level. The level of consistency preservation can be modified, and the routines themselves can be exchanged to easily incorporate more efficient algorithms for collection and checking. Figure 3 shows a sample consistency mapping table. Figure 4 illustrates the work of the consistency manager.

The approach taken here allows for a very high flexibility concerning the time at which consistency is imposed: In one extreme case, the collection routine is empty, and all checking is done at end-of-transaction time. In the other extreme, the collection routine has already enforced consistency. Thus, no work has to be done at end-of-transaction. This strategy may be changed, without modifications in the calls of the consistency manager or in the DBMS code.

The addition of the two routines for a newly defined consistency constraint can be done by the person responsible for DBMS management (often called database implementor) without much effort. The routines have to be implemented and compiled, the consistency mapping table has to be modified, and the system has to be linked anew. In appropriate environments, this modification can even be done during run-time of the system, if a dynamic binding is supported, and if the consistency manager offers an interactive interface to change the consistency mapping table.

6. Related Issues

It is near at hand to compare our approach to a trigger mechanism. Indeed, the consistency violation information collection can be implemented using an elaborate trigger concept (as described in [13], for example). However, a few problems remain unsolved when triggers are used:

Type of consistency	Level	Collect routine	Check routine
Cardinality restrictions	ADT	CA_Collect	CA_Check
Referential integrity access system	data system	RI_Collect	RI_Check
Referential integrity metadata	data system	RI_Collect	RI_Check
Storage structure consistency	access system	SS_Collect	SS_Check
Schema_Consistency	ADT	SC_Collect	SC_Check
• • •	• • •	• • •	• • •

Figure 3: A sample consistency mapping table

- It is not only necessary to trigger an action, but also to evaluate conditions in order to extract the information to be collected. The trigger mechanism therefore must support complex conditions.
- Not every trigger concept allows for the passing of parameters to the action part, which is absolutely necessary in our case.
- A common data structure has to be shared among various activations of trigger action parts. The corresponding synchronization mechanisms must be available.
- To also perform the consistency checking by triggers, the result of the triggered action must be returned to the consistency manager (or transaction manager). This, however, is not compatible with the concept of triggers.

The flexibility of a trigger mechanism, however, is gained with our approach as well.

The consistency constraints mentioned so far are in some sense system or model inherent (e.g. storage structure consistency or referential integrity). In advanced applications, however, it is often necessary to provide means to specify application dependent consistency constraints, even at different levels of the system. Usually, rules are used for this purpose, although an algorithmic specification is sometimes more adequate. This facility is easily integrated into our proposal by the introduction of a consistency constraint "user defined rules". The appropriate routines are rule checkers offered by the system. The programmer can use them at any level he wants, since the rule structure is uniform over all levels. These routines are then very similar to enriched trigger mechanisms [13]: In the event of "consistency may be violated" the condition part of the rules is evaluated and the appropriate action is fired if the condition is true. Hence, implementation can be done as efficiently as in trigger systems by using the same means as in these systems. Our approach, however, allows for deferring an action until the end of the transaction as well as for an algorithmic specifications of user defined consistency constraints.

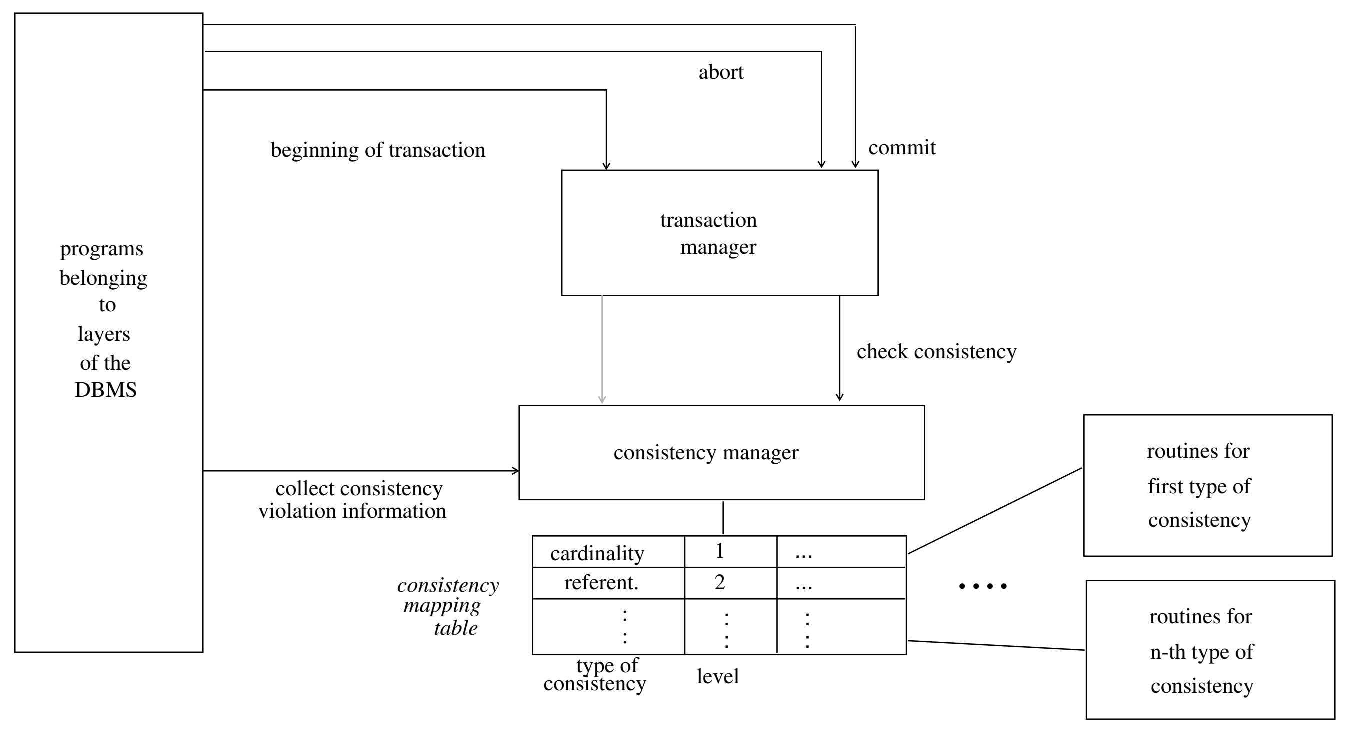

Figure 4: The components cooperating with the consistency manager

In design environments, the designer often wants to check for inconsistencies without committing a transaction (check before end of transaction). He can do so by calling the checking routine of the consistency manager himself (instead of calling it implicitly via the transaction manager as in the case of end of transaction).

[16] requires the facility to tolerate consistency violations which correspond to a real world's state (caused by an incorrect or incomplete modeling in the schema). We can include this facility in our approach by the integration of an additional routine **Tolerate_Inconsistency**, which tells the checking routine to ignore some inconsistencies. Nevertheless, we prefer correct modeling to explicitly allow for exceptions.

7. The Consistency Manager of PRIMA

We have implemented a consistency manager in the PRIMA project (prototype implementation of the MAD model [17, 18]). Its implementation can serve as an example for the principles discussed above. We consider only some of the layers of the PRIMA system in the following (cf. figure 5). The *application layer* offers an ADT oriented interface to the application programs. It is based on the *data system* which supports operations on complex objects (called *molecules* in the MAD model [19]). These are broken down to one-tuple operations executed by the *access system*. Similarly, the schema modification operations offered by the data system are broken down to operations executed by the *metadata manager*. A complex object (*molecule*) is represented by a graph consisting of basic objects (*atoms*) as nodes and references among them as edges. The references are represented by values of the so-called REFERENCE attributes. If a REFERENCE attribute of an atom *a* contains the IDENTIFIER value of atom *b*, then a REFERENCE attribute of *b* must contain the IDENTIFIER value of *a* (for details see [19]). Thus, a special kind of referential integrity must be preserved by each molecule operation. Since metadata are also organized as molecules, an analogous type of consistency has to be preserved in the metadata.

Schema modifications introduce a further type of consistency: For example, the deletion of an atom type requires the deletion of the corresponding REFERENCE attributes in all atom types that have references to the deleted one. This problem of schema consistency is distinct from the referential integrity problem.

The schema may contain cardinality restrictions for repeating group attributes which cannot be preserved by single molecule operations. Hence, the transaction at the application layer must guarantee these cardinality restrictions.

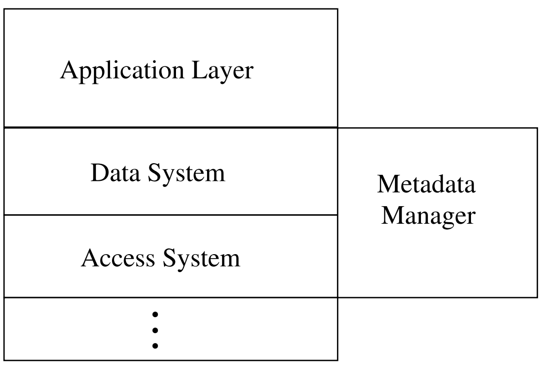

Figure 5: Some components of the PRIMA DBMS

Up to this point, we have identified four types of consistency constraints for PRIMA:

- referential integrity among atoms
- referential integrity in the metadata
- cardinality restrictions concerning attribute values
- schema consistency

Updates jeopardizing schema consistency take place in the data system, whereas for the other three types of consistency they take place in the access system. Figure 3 shows a consistency mapping table covering the four types of constraints mentioned above. As indicated, it is possible for two types of constraints to have the same two routines associated with them (as in the case of *referential integrity atoms* and *referential integrity metadata*). The following example illustrates the actions taken to maintain referential integrity at atom level for a sample operation at the ADT level. We assume a CAD application, where a three dimensional object o is to be cut by a plane resulting in object o' (figure 6).

In the MAD model, an object could be represented as molecule consisting of one atom representing the *object* itself, one atom for each *surface* of o, and one atom for each *edge* belonging to a surface of o (figure 7). The atoms describing *surfaces* and *objects*, and *surfaces* and *edges*, respectively, are connected to one another via references. The references established by pairs of REFERENCE attributes are depicted by connecting the circles representing atoms in figure 7.

The cut shown in figure 6 leads to three molecule operations, one deleting parts of o, one inserting new parts of o, and one updating parts of o. These are transformed into sets of access system operations, which include the deletion of surface ❸, the insertion of surface ❽, and possibly the update of surfaces ❶ ❷ ❹ ❺ (if there are any data changing - the references need not be changed in these surfaces, as figure 8 shows). The edges are handled in an according way. In figure 8, we show the actions at the access system level, grouped by their correspondence to the data system operations, and the actions taken by the consistency manager. To achieve simplicity of the example, we only consider referential integrity for atoms, although of course other types of consistency constrains are involved, too.

The figure depicts the "delete molecule" operation first. For this operation, the atom operations are shown. Each atom operation collects some consistency violation information. A (-) in front of the information indicates that the appropriate references have to be deleted in order to achieve consistent data. After an atom operation is finished, the consistency violation information is passed upward to the data system transaction. Figure 8 assumes an intelligent accumulation during the upward passing process, i.e. mutually neutralizing consistency violation information is deleted, and only those pieces of information are kept that really correspond to a consistency violation. After the last atom operation, at the end of the data system transaction, the consistency manager invokes the checking routine for referential integrity at atom level. Since referential integrity at atom level is a self-correcting consistency constraint, the routine forces the actions necessary to get consistent data. The necessary information has been constructed already in the collection phase due to the intelligent accumulation.

The second molecule operation needed for our example of figure 6 is the insertion of a molecule consisting of the insertion of a *surface* and its *edges*. Since there is no mutually neutralizing consistency violation information collected in this case, only the accumulated

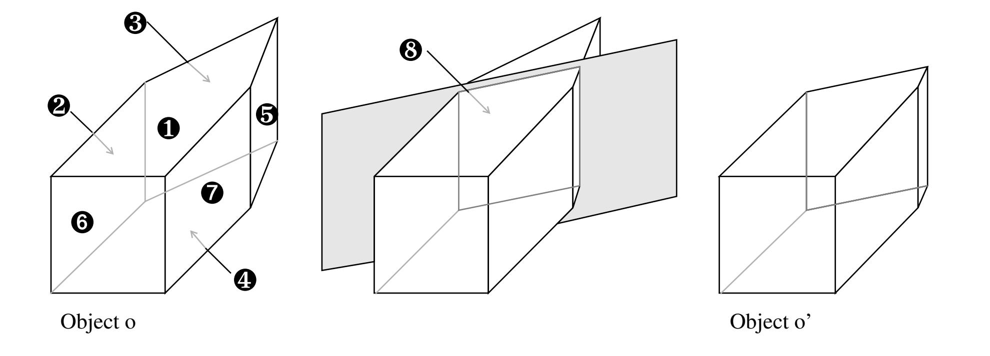

Figure 6: The three dimensional object o is to be cut by the shaded plane resulting in object o’.

information is shown. The (+) in front of the information collected indicates that those references have to be added to get a consistent database. When the checking routine adds the references, it also changes the newly inserted atoms representing the *edges* to enter the references to the new *surface*.

Now, it should be clear, why the other surfaces need not be explicitly changed, except if data (such as area) are represented: The references are corrected by the consistency manager, and the surfaces are updated implicitly.

8. Conclusions

In this paper, we have shown a way to integrate a nested view of consistency in a nested transaction concept. This allows a uniform view of “deferred” and “immediate” consistency constraints as well as of “global” and “local” consistency. Furthermore, the requirement for a flexible user reaction to consistency violations is fulfilled within the transaction concept. The introduction of a recording routine and a checking routine for each type of consistency constraint leads to an extensible architecture, allowing for the easy inclusion of new types of consistency constraints without modifications in the DBMS code. The concept also enables a flexible partitioning of work between the two routines tailored to the need of the corresponding consistency constraints.

[11] already had the idea to keep consistency violation log data during a transaction, and briefly sketched the integration of consistency into a nested transaction concept. Flexible reaction of the user, however, was not considered. [7] also differentiates checking and maintenance of constraints. The paper elaborates on the idea of “nested integrity transactions”, whereas the passing of protocol data as well as a flexible user reaction remain unconsidered. Both [11] and [7] do not mention the discrepancy between the level of consistency violation logging and the level of consistency guaranteeing. The concepts presented in this paper partly can be realized by the help of a elaborate trigger concept (as described in [13] for example): Triggers could be used to initiate the recording routines. For the checking of consistency at end of transaction, however, it is necessary to return the result of the check. Thus, triggers are inadequate to perform this task.

9. Acknowledgments

I would like to thank A. Sikeler, Th. Härder, and the anonymous referees for reading an earlier version of this papers and giving helpful comments.

References

[1] Moss, J.E.B: Nested Transactions and Reliable Distributed Computing, in: Proc. 2nd IEEE Symp. on Reliability in Distributed Software and Database Systems, Pittsburgh, 1982, pp. 33-39.

[2] Weikum, G.: Enhancing Concurrency in Layered Systems, in: Proc. 2nd Int. Workshop on High Performance Transaction Systems, Asilomar, 1987, LNCS 359, pp. 200-219.

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

[4] Eswaran, K.P., Grey, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database Sys-

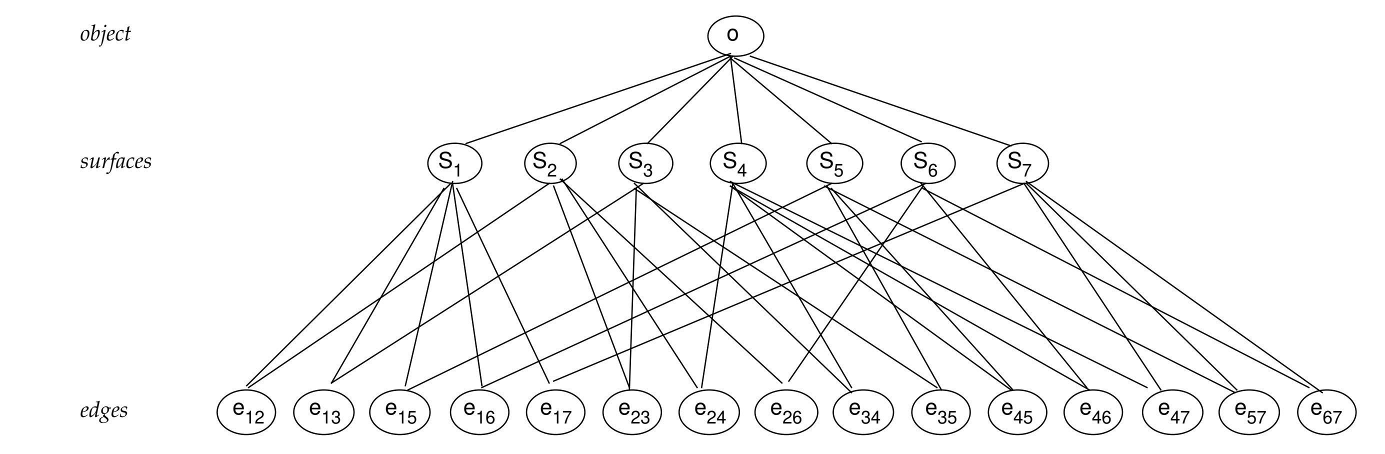

Figure 7: The graph established by the atoms of the object o and references among them.

The first molecule operation deletes S_3 and the belonging edges:

Beginning of a transaction at data system level

atom operation	consistency violation information *collected by the atom operation* and **intelligently accumulated**		
delete S_3	*collects information*	*(-)*	$e_{13}\rightarrow S_3$, $e_{23}\rightarrow S_3$, $e_{34}\rightarrow S_3$, $e_{35}\rightarrow S_3$, $o\rightarrow S_3$
	after upward passing	**(-)**	$e_{13}\rightarrow S_3$, $e_{23}\rightarrow S_3$, $e_{34}\rightarrow S_3$, $e_{35}\rightarrow S_3$, $o\rightarrow S_3$
delete e_{13}	*collects information*	*(-)*	$S_3\rightarrow e_{13}$, $S_1\rightarrow e_{13}$
	after upward passing	**(-)**	$S_1\rightarrow e_{13}$, $e_{23}\rightarrow S_3$, $e_{34}\rightarrow S_3$, $e_{35}\rightarrow S_3$, $o\rightarrow S_3$
delete e_{23}	*collects information*	*(-)*	$S_3\rightarrow e_{23}$, $S_2\rightarrow e_{23}$
	after upward passing	**(-)**	$S_1\rightarrow e_{13}$, $S_2\rightarrow e_{23}$, $e_{34}\rightarrow S_3$, $e_{35}\rightarrow S_3$, $o\rightarrow S_3$
delete e_{34}	*collects information*	*(-)*	$S_4\rightarrow e_{34}$ $S_3\rightarrow e_{34}$
	after upward passing	**(-)**	$S_1\rightarrow e_{13}$, $S_2\rightarrow e_{23}$, $S_4\rightarrow e_{34}$, $e_{35}\rightarrow S_3$, $o\rightarrow S_3$
delete e_{35}	*collects information*	*(-)*	$S_5\rightarrow e_{35}$ $S_3\rightarrow e_{35}$
	after upward passing	**(-)**	$S_1\rightarrow e_{13}$, $S_2\rightarrow e_{23}$, $S_4\rightarrow e_{34}$, $S_5\rightarrow e_{35}$, $o\rightarrow S_3$

The molecule operation is completed, the appropriate end of transaction causes the checking routine to be invoked. The checking routine deletes the following references:

$S_1\rightarrow e_{13}$, $S_2\rightarrow e_{23}$, $S_4\rightarrow e_{34}$, $S_5\rightarrow e_{35}$, $o\rightarrow S_3$

The second molecule operation inserts S_8 and the belonging edges:

Beginning of a transaction at data system level

atom operation	consistency violation information **intelligently accumulated**		
insert e_{18}	after upward passing	(+)	$S_1\rightarrow e_{18}$
insert e_{28}	after upward passing	(+)	$S_1\rightarrow e_{18}$, $S_2\rightarrow e_{28}$
insert e_{48}	after upward passing	(+)	$S_1\rightarrow e_{18}$, $S_2\rightarrow e_{28}$, $S_4\rightarrow e_{48}$
insert e_{58}	after upward passing	(+)	$S_1\rightarrow e_{18}$, $S_2\rightarrow e_{28}$, $S_4\rightarrow e_{48}$, $S_5\rightarrow e_{58}$
insert S_8	after upward passing	(+)	$S_1\rightarrow e_{18}$, $S_2\rightarrow e_{28}$, $S_4\rightarrow e_{48}$, $S_5\rightarrow e_{58}$, $e_{18}\rightarrow S_8$, $e_{28}\rightarrow S_8$, $e_{48}\rightarrow S_8$, $e_{58}\rightarrow S_8$, $o\rightarrow S_8$

The molecule operation is completed, the appropriate end of transaction causes the checking routine to be invoked. The checking routine inserts the following references:

$e_{18}\rightarrow S_8$, $e_{28}\rightarrow S_8$, $e_{48}\rightarrow S_8$, $e_{58}\rightarrow S_8$, $o\rightarrow S_8$

Figure 8: Collection of consistency violation information with intelligent accumulation

tem, in: Communications of the ACM, Vol 19, No 11, 1976, pp. 624-633.

[5] Gardarin, G: Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems, in: /20/, pp. 335-391.

[6] Schneider, F.B.: Ensuring Consistency in a Distributed Database System by Use of Distributed Semaphores, in: /20/, pp. 183-189.

[7] Eastman, C.M., Lafue, G.M.E.: Semantic Integrity in Design Databases, in: Encarnaçao, J., Krause, F.L. (eds.): File Structures and Data Bases for CAD, North Holland Publishing Company, 1982, pp. 45-54.

[8] Sinka, M.K.: Constraints: Consistency and Integrity, in: ACM SIGMOD Record Vol. 13, No. 2, 1983, pp. 60-63.

[9] Date, C.J.: An Introduction to Database Systems, 3rd ed., Addison-Wesley, 1981.

[10] Eswaran, K.P., Chamberlin, D.D.: Functional Specifications of a Subsystem for Data Base Integrity, in: Proc. 1st Int. Conf. on Very Large Data Bases, Framingham, 1975, pp. 48-68.

[11] Weber, W.: A Subsystem for the Maintenance of Semantic Integrity in Databases (in German: *Ein Subsystem zur Aufrechterhaltung der semantischen Integrität in Datenbanken)*, Ph.D. thesis, University Karlsruhe, Jochem Heizmann Verlag, Karlsruhe, 1981.

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

[13] Kotz, A.M., Dittrich, K.R., Mülle, J.A.: Supporting Semantic Rules by a Generalized Event/Trigger Mechanism, in: Advances in Database Technology - EDBT '88 (Int. Conf. on Extending Database Technology), Venice, Italy, 1988, pp 76-91.

[14] Stonebraker, M.R.: Implementation of Integrity Constraints and Views by Query Modification, in: ACM SIGMOD Proc. Int. Conf. on Management of Data, San Jose, California, 1975, pp. 65-78.

[15] Hammer, M., Sarin, S.K.: Efficient Monitoring of Database Assertions, in: ACM SIG MOD Proc. Int. Conf. on Management of Data, Austin, Texas, 1978, pp. 159-168.

[16] Borgida, A., Mitchell, T.M., Williamson, K.: Learning Improved Integrity Constraints and Schemas From Exceptions in Databases and Knowledge Bases, in: Brodie, M.L., Mylopoulos, J. (eds.): On Knowledge Base Management Systems, Springer 1986, pp. 259-286.

[17] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. 13th Int. Conf. on Very Large Data Bases, 1987, pp. 433-422.

[18] Härder, T. (ed.): The PRIMA Project - Design and Implementation of a Non-Standard Database System, Sonderforschungsbereich 124, Report 26/88, University Kaisers lautern, 1988.

[19] Mitschang, B: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. 2nd Int. Conf. on Expert Database Systems (ed. L. Kerschberg), 1988, pp. 33-50.

[20] Delobel, C, Litwin, W. (eds): Distributed Databases, North Holland, 1980.