Unland, Rainer

Working Paper

Optimistic concurrency control revisited

Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 30

Provided in Cooperation with:
University of Münster, Department of Information Systems

Suggested Citation: Unland, Rainer (1994) : Optimistic concurrency control revisited, Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 30, Westfälische Wilhelms-Universität Münster, Institut für Wirtschaftsinformatik, Münster

This Version is available at:
https://hdl.handle.net/10419/59331

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgeber: Prof. Dr. J. Becker, Prof. Dr. H. L. Grob, Prof. Dr. K. Kurbel,
Prof. Dr. U. Müller-Funk, Prof. Dr. R. Unland, Prof. Dr. G. Vossen

Arbeitsbericht Nr. 31

Optimistic Concurrency Control Revisited

Rainer Unland

Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster,
Grevener Str. 91, 48159 Münster, Tel. (0251) 83-9750, Fax (0251) 83-9754
März 1994

Contents

Abstract

Several years ago optimistic concurrency control gained much attention in the database community. However, two-phase locking was already well established, especially in the relational database market. Concerning traditional database systems most developers felt that pessimistic concurrency control might not be the best solution for concurrency control, but, a well-known and accepted one. With the work on new generation database systems, however, there has been a revival of optimistic concurrency control (at least a partial one). This paper will reconsider optimistic concurrency control. It will lay bare the shortcomings of the original approach and present some major improvements. Moreover, several techniques will be presented which especially support read transactions with the consequence that the number of backups can be decreased substantially. Finally, a general solution for the starvation problem is presented. The solution is perfectly consistent with the underlying optimistic approach.

1 Introduction

Several years ago optimistic concurrency control gained much attention in the database community. However, two-phase locking was already well established, especially in the relational database market. Concerning traditional database systems most developers felt that pessimistic concurrency control might not be the best solution for concurrency control, but a well-known and accepted one. With the work on new generation database systems, however, there has been a revival of optimistic concurrency control (at least a partial one). This can be seen by the fact that optimistic concurrency control found its way in at least one commercial (object-oriented) database system, namely GemStone ([Maie89]). There are several application areas for which optimistic concurrency control may be of interest. Among them are:

- **Real-time transaction processing (high data contention)**
  Real-time Applications require a fast response and a high degree of concurrency. Since optimistic concurrency control has the properties of non-blocking and deadlock freedom, they are especially attractive to real-time transaction processing (cf. [HSRT91], [HaCL90],). Let us, for example, consider an airline reservation system. In such a system a high degree of concurrency is essential. In case of a conflict, the conflict must not necessarily be serious. For example, a typical transaction, checking the availability of seats on a particular flight, needs only to know whether the current value of available seats is non-zero. As long as there are still seats available available, a possible conflict between two transactions can be neglected (cf. [JaSh92]).

- **Cooperative environments**
  Cooperative environments usually require a concurrent work on objects. However, nevertheless a possible conflict must be detected and brought to the knowledge of the corresponding users or application. They can decide on their own how to react, e.g. either by initiating a compensating action or by starting user or application defined repair actions.

Both examples have in common that they require a concurrent, often even non-serializable work on data. However, in case of a conflict this conflict must be detected. Here, optimistic methods seem to be quite appropriate since they allow non-serializable work on data, however, discover every conflict. Of course, in situation like the above, optimistic concurrency control should not automatically rollback transaction but leave it to the application to trigger the appropriate reaction.

This paper will reconsider optimistic concurrency control from a general point of view; i.e., it will not discuss the use and adaptation of optimistic concurrency control for special application areas. Instead, it will present major improvements and solutions for most weak points of the original approach to optimistic concurrency control.

The remainder of the paper is organized as follows. In section 2 the original approach to optimistic concurrency control will be introduced briefly. Section 3 will analyse the original approach to lay bare its shortcomings. As a consequence of this analysis several improved validation schemes will be presented in section 4. These solutions will not only eliminate the weakness of the original approach but, additionally, will reduce the overall cost for validation. In section 5, we will propose a validation scheme, which especially supports read transactions. With this scheme every read transaction will survive if there is a chance to do so. In section 6 it will be discussed how read transaction can run without any consideration of concurrency control at all. This will be achieved by the integration of a version approach. Section 7 concentrates on the starvation problem and presents several solutions which are entirely based on the optimistic approach. Finally, section 8 will conclude this paper.

2 Original approach to optimistic concurrency control

We give a brief description of the basic optimistic concurrency control scheme as proposed in [KuRo79]. A transaction consists of three phases: *a read phase, a validation phase* and *a write phase*. In the read phase the required objects are read from the database, write operations are performed on local copies of the database objects. In the validation phase a check for serializability is performed. If validation is successful, the objects modified by the transaction are written into the database (write phase), otherwise the transaction is restarted. Validation and write form a critical section.

Validation is performed as follows: For each transaction T_i the system keeps track of the set of objects read from the database (RS_i) and of the set of objects written (WS_i). If validation is successful, the transaction is assigned a unique transaction number TNR_i. For this purpose a global transaction counter TNC is maintained. Let TNR_{start} be the highest transaction number at the start of transaction T_j, and let TNR_{finish} be the highest transaction number at the start of validation. Then T_j performs the following check:

```
(VAL) <valid := true;
        for TNR from TNR_start+1 to TNR_finish do

            if RS_j ∩ WS_i ≠ ∅ then valid := false;

        if valid then
        begin

            (write);
            TNR_j := TNC;
            TNC  := TNC+1

        end>

       if not valid then (backup);

<>     marks the critical section
```

Algorithm 1: Validation as in [KuRo79]

This scheme guarantees a serialization in the order of the transaction numbers (equal to the order of commit).

In the next chapter it will be shown that this approach is too pessimistic.

3 Shortcomings of the original approach

There are some simulation studies (see, e.g., [Agra83], [AgCL85], [AgCL87], [AuPS84], [Bhar82], [Bhar80], [Care83], [FrRo85], [HSRT91], [HuSt90], [KeTe84], [MeNa82], [PeRe83], [TaGS84]) which are quite positive for optimistic methods, though there is no clear advantage over locking. Greater acceptance can only be achieved if some serious drawbacks of the original approach of Kung and Robinson [KuRo79] can be overcome.

The following issues are essential:

1. The degree of potential parallelism of optimistic methods is high. However, the same is true for the risk of being restarted unnecessarily. It would be desirable to avoid restart in all cases where detected conflict actually does not endanger serializability.

2. Since long transaction run for a longer time than transactions of average size they have an increased risk of being subject of restart. However, long transactions should have similar chances of committing than short ones.

3. Since optimistic concurrency control relies on rollback as the means of synchronization transactions might be restarted repeatedly. In worse cases the same transaction might be the victim of rollback again and again. This phenomenon is known as the starvation problem and, of course, must be prevented.

4. Optimistic concurrency control is considered to be especially favorable for query intensive applications. Therefore, validation for read transactions should be flexible enough to allow each transaction a normal termination whenever there is a chance to do so.

5. A main weakness of the original validation scheme is its clumsy definition of conflict. As a consequence transactions may be restarted unnecessarily, as figure 1 shows:

   Transaction T_h writes during the read phases of T_i and T_j and thus has to be considered in their validation phases. Using the validation scheme of [KuRo79], both transactions have to be restarted. If we take a closer look at the above scenario the following becomes clear:

   ☞ T_i reads an object that will later be written by T_h. Serialization in commit order is not possible. Hence this conflict is "serious": It signals a non-serializable situation. Note that serialization in the reverse order of commit is not possible in general, see [PrSU82].

   ☞ However, the conflict between T_j and T_h is not "serious", since the conflict due to *y* simply expresses the serialization constraint $T_h \rightarrow T_j$. Restart of T_j is unnecessary.

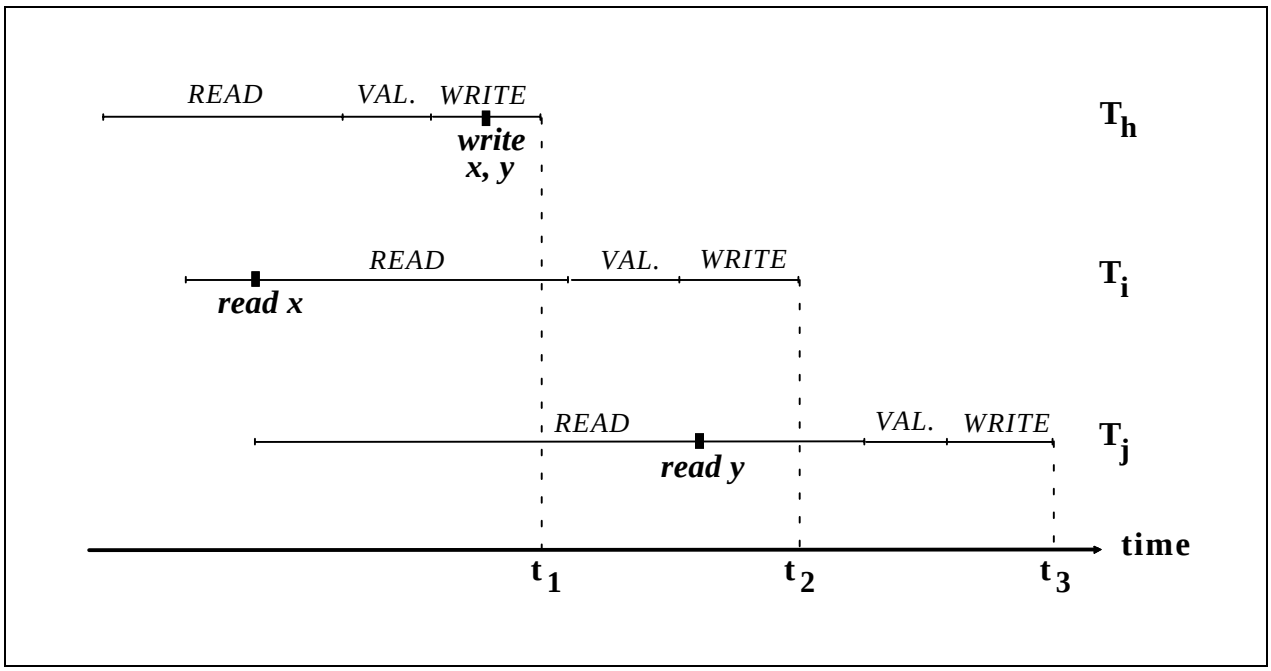


Figure 1: Validation as in [KuRo79]

The outlined situation is especially unfavorable, since transactions may be restarted because of conflict with T_h even if their first access to the database was after the write of T_h. Or, to come back to our example applications in the introduction, a non-existent conflict is signalled to the user or application as a serious one.

Observation:

> Every conflict between an update-transaction T_h and a concurrent transaction T_j is **serious** for serialization in commit-order if the conflict occurs before the end of the write-phase of T_h. It is **non-serious** if it occurs afterwards. As will be shown later, a serious conflict must not in each case lead to a rollback. However, it must lead to a rollback if serialization in commit-order is assumed. Therefore, we will call such a conflict **commit-serious** (or **c-serious** for short).

In the following we propose several improved validation schemes whose common feature is that they distinguish between "c-serious" and "non-c-serious" conflicts, and thus substantially reduce the risk of restart.

4 Improvement of validation

4.1 Solution 1: EOT marker

The following idea is a simple but effective solution for recognizing certain "non-c-serious" conflicts: At each EOT of an update-transaction T_j every parallel transaction T_i takes a note of the termination of T_j in its read-set. Provided that the read-set of T_i is ordered in the sequence of its actions on objects, the following holds: During validation against T_j T_i has to consider only the part from the beginning of the read-set up to the point where the EOT of T_j is marked. All actions done after the commit of T_j can produce only "non-c-serious" conflicts (see figure 1).

Example 1:

EOT_l denotes the EOT marker of T_l in a read-set. Assume T_i has the following read-set:

$(EOT_l, x, y, EOT_m, z, EOT_n, v, w, EOT_p)$

In its validation T_i has to perform the following tests:

1. no test against T_l
2. $WS_m \cap (x, y)$
3. $WS_n \cap (x, y, z)$
4. $WS_p \cap (x, y, z, v, w)$

On the basis of the original validation scheme all validation tests would have been performed against the read-set of test 4. of above.

This proposal is the basis for the modified validation scheme for read-transactions (see chapter 5).

4.2 *Solution 2:* Snapshot validation

The following validation scheme is called snapshot validation, because validation does not consider terminated transactions but a snapshot of the actual state of all transactions; i.e. it considers all concurrent transactions that are in their read-phase. This scheme makes use of the fact that a c-serious conflict of an update-transaction T_h with a concurrent transaction T_j can

only occur before the end of the write-phase of T_h (as was shown in chapter 3). Hence, any transaction can check its safety with respect to T_h as soon as T_h has committed.

Consider Figure 2. After the commit of T_h, time t_1, each concurrent transaction T_m (m = i, j, k) performs the check $RS_m(t_1) \cap WS_h$, where $RS_m(t_1)$ is the read set at time t_1. If a conflict is detected, it is a c-serious one with the consequence that T_m must be restarted.

In the original approach validation is performed against the write set of all transactions that have committed during the run-time of T_h, whereas now, at each commit of a transaction T_h, all concurrent transactions validate their current read set against T_h's write-set.

There is no longer one single validation phase for a given transaction. Instead validation is performed constantly during T_h's run-time. Nevertheless, the overall number of validation points is usually less than in the original approach (each write-transaction causes one while a read-transaction causes none).

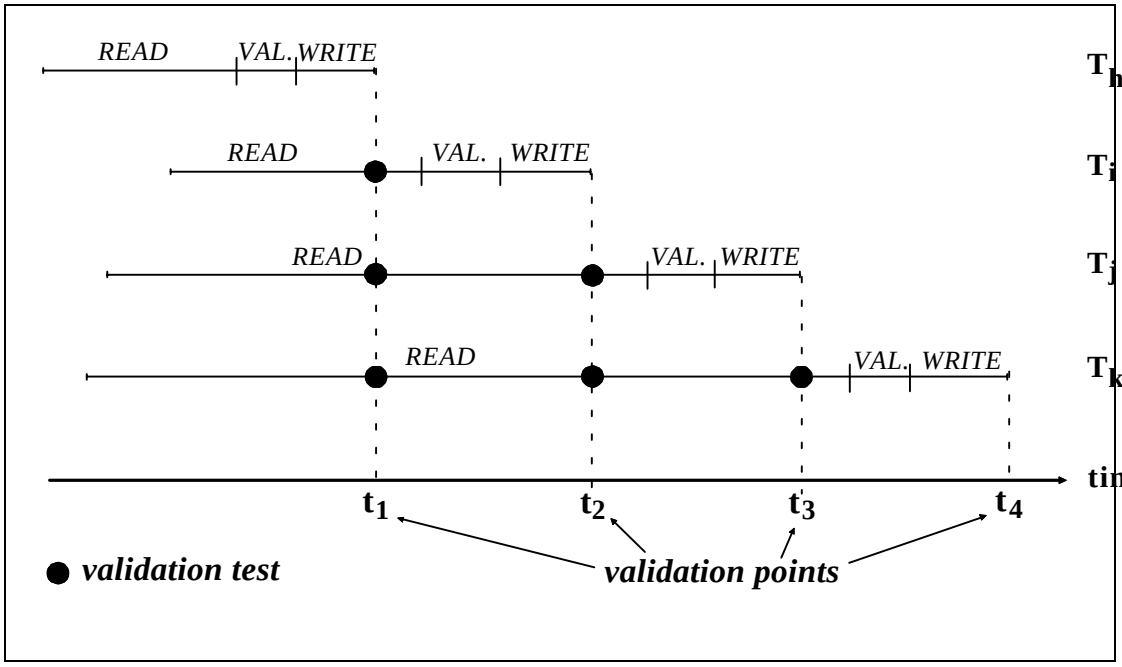


Figure 2: Snapshot validation

There are several remarkable advantages of the snapshot validation:

- Only c-serious conflicts cause a back up of a transaction.
- The read-sets to be considered in the validation phase are considerably smaller in the average.

- Conflicts are detected earlier. A transaction no longer runs to its end to detect that it has to be restarted.

- The write set of a transaction must no be kept until all concurrent transactions are terminated; it can be released immediately after the validation of all concurrent transactions. Because this validation is either done during the validation phase of T_h or just after the termination of T_h (see below), write sets have to be stored for considerably shorter periods.

We can distinguish between two types of snapshot validation schemes, namely snapshot validation with or without critical section. In the scheme with critical section, validation of concurrent transactions is done during the validation phase of the terminating transaction. Therefore, in case of a conflict there is a choice as to which of the conflicting transactions should be backed up. So, this alternative includes a solution for the starvation problem. A drawback of this proposal is the use of a long critical section, which start with the validation phase of a terminating transaction and end with its EOT. During this section the whole database is locked for all concurrent transactions.

Snapshot validation without critical section avoids long critical sections since all validation tests of concurrent transactions are performed immediately after the EOT of the terminating transaction. Clearly, now there is no freedom to choose the transaction that is to be backed up. Thus, starvation may again be a problem. Therefore, we propose, that this solution should be used in combination with the (starvation avoiding) algorithm presented in section 7.

4.3 Snapshot validation with critical section

The following scheme assumes that validation and write form one continuos critical section during which no transaction is allowed to access the database. Therefore, the read sets remain unchanged. In this case, it is possible to perform the snapshot-validation before the write phase of the terminating transaction.

Consider figure 3. Transactions T_i, T_j and T_k are validated against T_h in the interval t_1 to t_2. Conflicts are detected before T_h writes its updates to the database. Since the validating transaction T_h validates against all running transactions a transaction counter is no longer necessary; i.e., we no longer need the transaction counter to identify the transactions against which T_h has to validate.

This validation scheme opens up new possibilities to resolve conflicts: in the original approach a conflict is always resolved by restarting the validating transaction; now each of the two conflicting transactions can be restarted. As a result the following strategies for conflict resolution can be implemented:

1. In case of conflicting update transactions the decision of whom to restart can be based on explicit priorities of transactions or on other criteria such as processing time used so far.

   Note that strategies of this type may solve the problem of long transactions as well as the starvation problem: one simply has to use as a criterion the accumulated processing time of a transaction (including all restarts). This automatically gives priority to long transactions and to transactions which have been restarted. A similar effect can be reached by assigning a special priority to a transaction which has to come to its end safely.

2. If a read transaction conflicts with a write transaction, one can delay the commit of the writer until the reader has terminated. Read transactions would not have to do validation at all (see also [Schl81]). However, this approach can imply serious disadvantages for writers.

In addition to the advantages already mentioned the outlined validation scheme includes simple and efficient solutions for the problems of long transactions and starvation. The price to be paid is a long critical section which may result in substantial blocking of concurrent transactions (see algorithm 2). The scheme presented in the next section avoids this disadvantages.

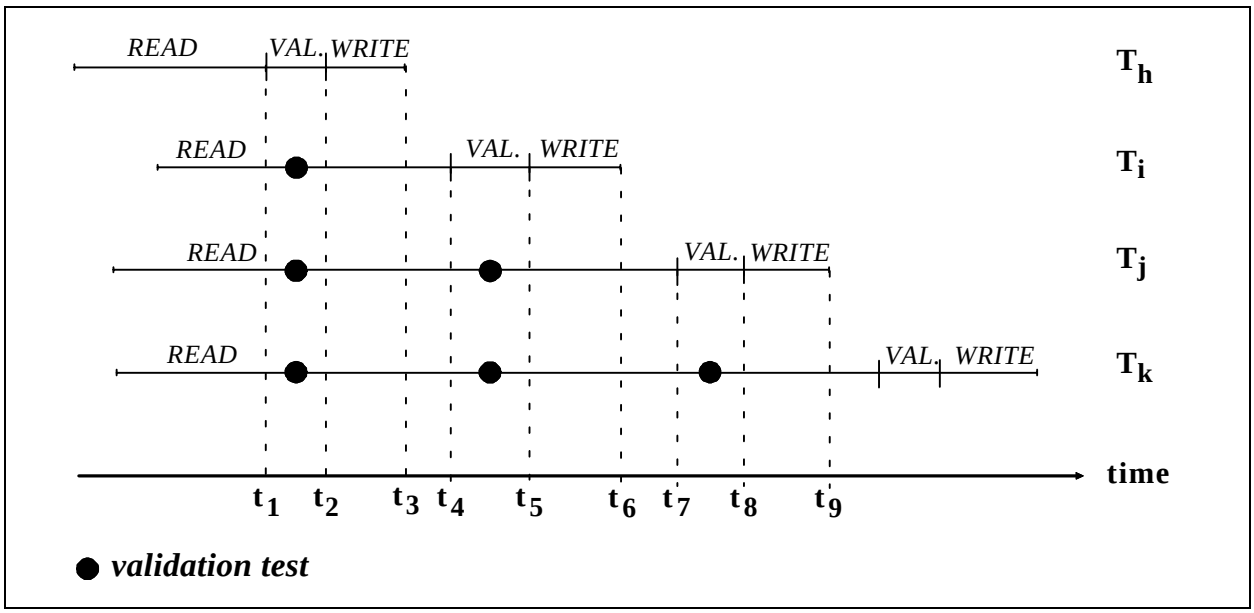


Figure 3: Snapshot validation *with* critical section

(**VAL**) <valid := **true**;
for all (transactions T_i which are still running and have not yet started validation)

if $RS_j \cap WS_i \neq \emptyset$ **then** valid := **false**;

if valid **then** (write);>
if not valid **then** (rollback one of the conflicting transactions);

<> *marks the critical section*
T_j *validating transaction*

Algorithm 2: Snapshot validation *with* critical section

4.4 Snapshot validation without critical section

Two transactions T_i and T_j are correctly synchronized in the serialization order $T_i \rightarrow T_j$, if the following holds:

(SR) (1) T_j does not read or write objects written by T_i
(2) T_i terminates its read phase before T_j terminates its read phase

We assume that updates of transactions are done on local copies of the database objects. Furthermore, we require an object to be read from the database before it can be modified. Therefore, for all transactions $WS \subseteq RS$ holds. As a consequence, the test $RS_j \cap WS_i$ includes the test $WS_j \cap WS_i$. The test $RS_j \cap WS_i$ has to be executed at a point of time where the read set of T_j contains all objects which could produce a c-serious conflict to T_i (remember that the write phase of T_i is not a critical section). The earliest point for this test is

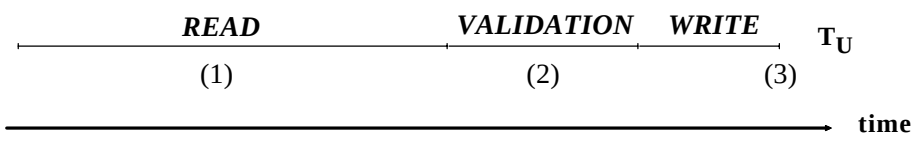
- ☞ either the begin of validation of T_j, if T_j starts validation before the end of T_i's write
- ☞ or the end of T_i's write phase, if T_j is still in its read phase at this time.

Condition (2) is satisfied if transaction numbers are assigned at the beginning of validation, and if $TNR_i < TNR_j$.

An update transaction T_i now has the structure given in figure 4.

During the read phase T_i validates against all transactions terminating in this interval. At the start of validation T_i is assigned its transaction number TNR_i. Thus, all concurrent transactions that validate later will have to validate against T_i. T_i has to check whether there are transactions with transaction number smaller than TNR_i that have not yet terminated. No validation has been done against these transactions, so a check has to be performed now. After these steps T_i can commit its updates provided the checks were positive. After the commit concurrent transactions validate against T_i. Figure 5 gives an example:

After the termination of T_h transactions T_i, T_j and T_k validate against T_h. Later, at the beginning of T_i's final validation, T_j has not yet terminated. As $TNR_j < TNR_i$, T_i has to validate against T_j now.



(1) validation against transactions terminated in this phase
(2) assignment of transaction number, validation against transactions with smaller transaction number which have not yet terminated
(3) T_i's termination is made public

Figure 4: Structure of an update transactions

This validation scheme is highly flexible and allows maximal parallelism as the necessary critical section is reduced to the access of the transaction counter. There is no restriction as to the sequence of the commits. Some simulations [PeRe83], [AuPS84] show, that snapshot validation behaves considerably better than the original approach and that it is a good candidate for replacing locking schemes in a variety of applications.

In [AgCL85] the original approach of optimistic concurrency control is compared to the locking scheme. As in the above mentioned simulations, the original approach is inferior to locking in some applications. However, the differences between both methods are small, so that it is very likely that snapshot validation will be superior to locking in most applications.

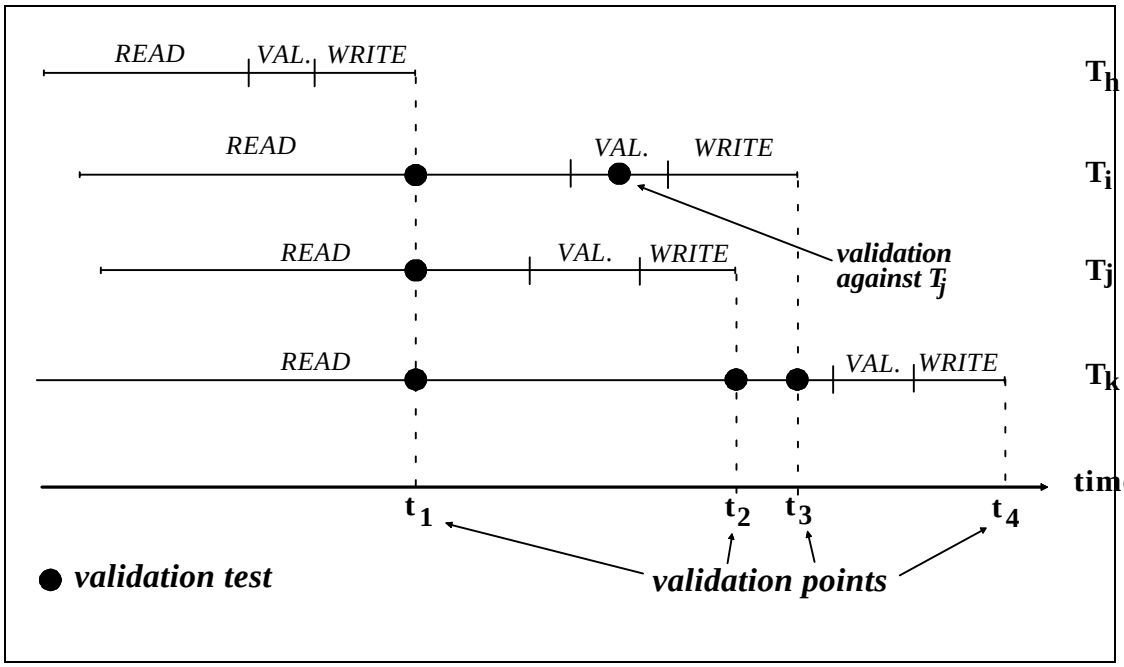


Figure 5: Snapshot validation *without* critical section

(**VAL**) valid := **true**;
<TNR_j := TNC;

TNC := TNC+1>

for all (transactions T_j with smaller TNR which have not yet terminated)

if $RS_i \cap WS_j \neq \emptyset$ **then** valid := **false**;

if not valid **then** (backup T_i);

(write);
for all (transactions T_j which are still running and have not yet started validation)

if $WS_i \cap RS_j^{current} \neq \emptyset$ **then** backup (T_j);

termination of T_j is made public;

<> *marks the critical section (only modification of transaction counter)*
T_j *validating transaction*
$RS_j^{current}$ *current read-set of transaction T_j*

Algorithm 2: Snapshot validation *without* critical section

A drawback of this solution is that the simplicity of the solution for the support of long transactions and of the starvation problem is lost. However, in section 6 we will discuss a general solution for the starvation problem.

Another possibility to approach these problems is to combine the two versions of snapshot validation. As a rule, transactions behave according to the validation scheme without critical section. However, a long transaction or a transaction which has been restarted several times may be given priority: each terminating transaction T_i must validate against concurrent transactions with priority. In case of conflict T_i is restarted. The transaction with priority must be preempted during this validation. If conflict is possible between two transactions with priority, then either priority levels or other standard decision mechanisms are required.

5 A validation scheme for read transactions

In this chapter we introduce a validation algorithm that especially improves the validation scheme for read transactions.

Usually concurrency control tries to place a transaction at the end of the serial schedule existing so far. If this is not possible, the transaction is backed up. In contrast to this usual approach the new algorithm does not back up a transaction in this case but tries to place it at some other point in the serial schedule, thus making backup unnecessary in many cases (see figure 6).

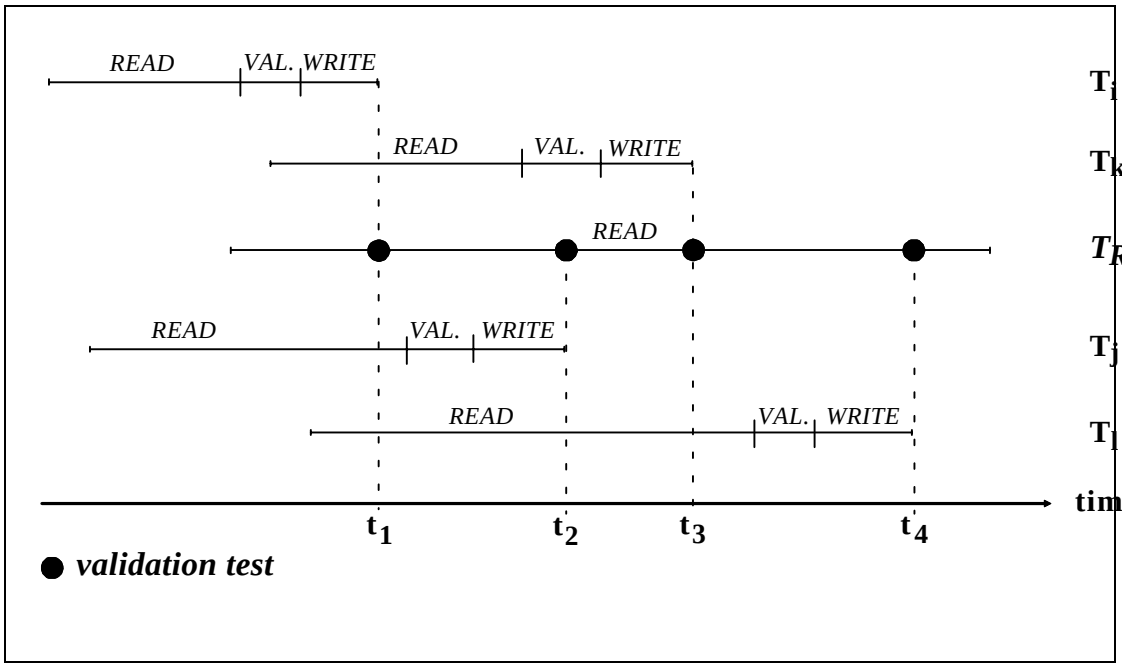


Figure 6: Validation scheme for read transactions

In the above scenario T_R is a read transaction while all other transactions are update transactions. If all transactions read their validation number in the sequence of their termination snapshot validation would try to produce the following serial schedule:

(SS) $T_i \rightarrow T_j \rightarrow T_k \rightarrow T_l \rightarrow T_R$

Assume there is a conflict between T_k and T_R. This means that T_R read an object before t_3 that was written by T_k (note: T_R has not read the version written by T_k since write takes only place at time t_3). Snapshot validation (or any other optimistic validation scheme) would back up T_R, because the serial schedule (**SS**) is no longer possible. However, although the conflict prevents the serial schedule (**SS**), this does not mean that the serialization criterion is violated. T_R has

executed validation against T_i and T_j at time t_1 and t_2, respectively, and no conflict was detected. Therefore it would be possible to place T_R between T_j and T_k in the serial schedule, if the following holds:

☞ T_R does not read any object written by T_k or any other transaction located after T_i in the serial schedule.

Let T_R be a read transaction and T_U a concurrent update-transaction. Then RS^{BU} indicates that part of T_R's read-set from the beginning of T_R up to the point where T_U starts its write-phase. Analogously, RS^{AU} indicates the part of the read-set of T_R starting with the write-phase of T_U and ending with the end of the read-phase of T_R (see figure 7):

The terminating read transaction T_R can be placed at a point x in the serial schedule if the following is true:

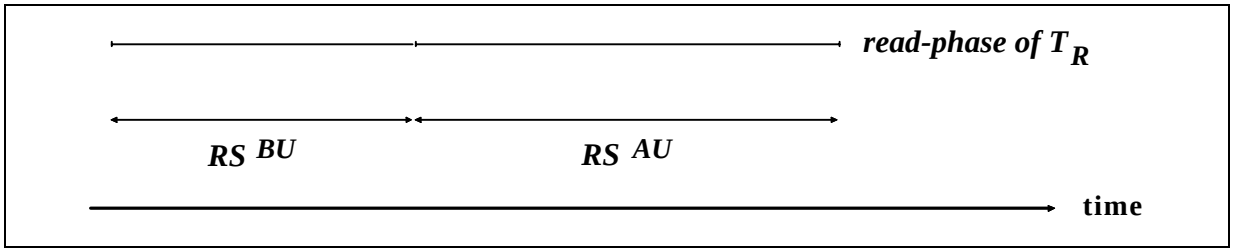


Figure 7: Read-set of T_R, divided by T_U

(a) $WS_U \cap RS^{BU} = \varnothing$ for all concurrent update transactions T_U standing before x in the serial schedule.

(b) $WS_U \cap RS^{AU} = \varnothing$ for all concurrent update transactions T_U standing after x in the serial schedule.

Condition (a) is satisfied by the usual test executed by snapshot validation because validation of a running transaction T_R against a terminating update transaction T_U is done immediately after the EOT of T_U with the read-set of T_R existing so far. This test is the usual test for the new validation scheme, too. The first time a conflict occurs, this validation scheme is no longer usable. Condition (b) has to be checked from this time.

To fulfil condition (b) a method similar to the proposal of chapter 4.1 is introduced:

Every read transaction T_R records the begin of the write phase of every terminating update transaction T_U in its read-set (say, BOW_U marker). Of course, the sequence of actions has to

be preserved in the read-set. Because the complete read-set of T_R is not available as long as T_R is in its read phase from now on validation against all update transactions has to be done at the end of the read phase of T_R. Therefore the write-sets of all terminating update transactions have to be preserved. In its validation phase T_R has to consider only the part from the BOW_U marker up to the end of the read-set (see condition (b)).

In general the method works as follows:

As long as no conflict is detected, every read transaction T_R works as usual, for example according to the validation scheme of the snapshot validation. Additionally, the begin of the write phase of every terminating update transaction (BOW marker) is recorded in the read-set of T_R. If the first conflict occurs T_R changes its validation technique, such that:

1. the read-set existing so far can be deleted up to the BOW entry of the conflicting transaction.

2. in contrast to the rules of the snapshot validation from now on T_R does not validate immediately after the EOT of a concurrent update transaction T_U but at its end. For this reason, the write-set of T_U is stored.

3. Before terminating T_R performs the following test against all transactions recorded in its read-set:

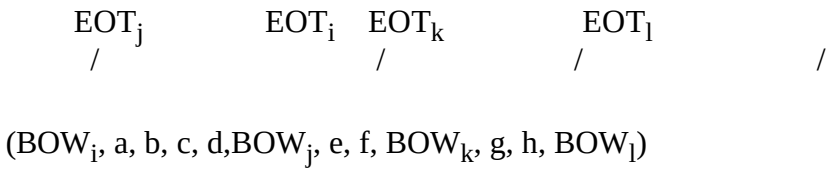
$$WS_U \cap RS^{AU} = \varnothing$$

If no conflict occurs, T_R can successfully terminate. In the serial schedule T_R is placed just before the transaction causing the change of validation technique.

Example 2:

The basis of this example is the scenario of figure 6.

Assume T_R has the following read-set:



As T_k is the conflicting transaction the following tests will be performed:

usual validation:

against T_i : none, because the read-set is empty
against T_j : $RS_R \cap$ (a, b, c, d)
against T_k : $RS_R \cap$ (a, b, c, d, e, f), a conflict occurs,

therefore:

change of validation policy (validation tests are done after the read phase of T_R)
against T_k: $RS_R \cap$ (g, h)
against T_l : none, because T_R has not read an object after the begin of the write phase of T_l.

The considered algorithm allows read transactions to survive if there is a chance. We do not know any other algorithm which takes a look at the serial schedule existing so far in order to place a conflicting transaction at a possible point in this schedule. In comparison to the original approach validation is quite short.

6 Multiversion optimistic concurrency control

Optimistic methods are supposed to be especially favorable in query-intensive applications. Therefore, it would be useful to separate the concurrency control for readers and writers in a way that readers run without any risk of restart. A possible solution has been indicated earlier: in case of conflict between reader and updater the write phase of the updater is delayed. However, it is obvious that this scheme may produce intolerably long delays for updaters, and also may increase the risk of restart for the updater, since, while it waits, other updaters may terminate.

A solution that avoids conflict between read- and write-transactions in the context of two-phase locking is extend two-phase locking with versioning (cf. [CFLN82], [DuBo82], [PaKa84]). Under multiversion two-phase locking, prior versions of objects are retained to allow queries to run against past transaction-consistent database states. The presence of versions allows queries to serialize bafore all concurrent update transactions, and thus queries and update transactions do not conflict. Zhis scheme can be adapted to optimistic concurrency control. By the use of versions a reader always can get a consistent view of the database. In our proposal, a reader works on the database version that existed at the start of the

transaction. In this scheme there is no overhead for concurrency control for readers. Moreover, update transactions do not need to consider concurrent readers any longer.

To implement a version concept, it seems to be advantageous to use pages as the basic means of concurrency control (there are also other reasons to base optimistic concurrency control on pages, which we cannot discuss here. Some interesting arguments concerning this issue are given in [Härd84]).

The version concept can be outlined as follows: Update transactions work on copies of pages. At commit time the modified pages are marked with the transaction number TNR. The old versions of the modified pages are kept. Update transactions work on the current versions, whereas readers work on versions which were current at their start; modifications which occur during a reader's life are not considered. To accomplish this behaviour, a read transaction T_R reads at its beginning the value $TNCR_R$ of a counter TNCR; TNCR indicates the transaction number of the latest terminated update transaction. At each access to a page T_R tests whether the stamp of the page is less or equal to $TNCR_R$; in this case the page was not modified since the start of T_R. If the stamp is greater than $TNCR_R$, this page was modified in the meantime and T_R would get an inconsistent view of the database. To avoid this T_R then accesses the older version of this page. Note that in some cases more than one old page version may exist: T_R then has to access the page with the highest stamp less than $TNCR_R$. Old versions must exist as long as a read transaction is active the $TNCR_R$ of which is smaller or equal to the stamp of the page.

In the case of snapshot-validation with critical section the TNCR corresponds to the TNC, which means that a separate counter for the TNCR is not required. More complicated is the case of snapshot validation without critical section. There, the sequence of termination of transactions does not necessarily correspond to the sequence of transaction numbers, so that there may be gaps in the list of terminated transactions. For instance, the transactions with the numbers 100, 102 and 103 may be terminated while T_{101} and T_{104} are still running. A read transaction starting at this time must not work with TNCR 103, since this can imply that possible conflicts with T_{101} remain undetected. There are two solutions for this problem:

1. Enforce termination of update transactions in the sequence of their transaction numbers.

2. Use the last TNCR value before which there is no gap in the list of terminated transactions. In the above example this value would be 100. The disadvantage of this approach is that updates of terminated update transactions may stay invisible even for read transactions that started after the termination of the update transaction.

Summary

Table 1 summarizes the different validation techniques:

	starvation problem	**critical section**	**read transactions**
VAL^{cs}	*solved*	*long*	*no special treatment*
$VAL^{\neg cs}$	*¬solved*	*extremely short*	*no special treatment*
VAL_{cs}^{read}	*solved*	*long*	*will survive whenever there is a chance*
$VAL_{\neg cs}^{read}$	*¬solved*	*extremely short*	*will survive whenever there is a chance*
$VAL_{cs}^{version}$	*solved*	*long*	*will survive in any case*
$VAL_{\neg cs}^{version}$	*¬solved*	*extremely short*	*will survive in any case*

VAL^{cs} : *snapshot validation with critical section*

$VAL^{\neg cs}$: *snapshot validation without critical section*

VAL_{cs}^{read} : *snapshot validation for read transactions with critical section for update trans.*

$VAL_{\neg cs}^{read}$: *snapshot validation for read transactions without critical section for update trans.*

$VAL_{cs}^{version}$: *multiversion snapshot validation with critical section*

$VAL_{\neg cs}^{version}$: *multiversion snapshot validation without critical section*

Table 1: Characteristics of the different validation schemes

7 Substitute transactions: a general solution for the starvation problem

Since in the optimistic approach concurrency control is based on backing up transactions which cannot be placed at the end of the serial schedule existing so far, a transaction may be restarted again and again. This problem is usually called starvation. The risk of starvation is the greater the longer the transaction is, long transactions may especially be killed by short ones. A very restrictive solution for the starvation problem is suggested in [KuRo79]: after a transaction has been restarted a certain number of times, in principle the whole database is locked for it. It is

obvious that such a solution is not acceptable for databases with a high degree of parallel usage.

The solution

We present a solution for the starvation problem which is entirely based on the optimistic approach.

Starvation occurs if, during validation, a transaction T_i repeatedly finds a T_j such that $RS_i \cap WS_j \neq \emptyset$. After a certain number of trials of T_i we have to make sure that T_i is not backed up again. We propose the following solution:

After a certain number of trials for transaction T_i a substitute transaction T^S_i is created. T^S_i possesses the read- and write-sets of T_i. The purpose of T^S_i is to prevent other transactions T_j from validating positively if they are in conflict with T_i, as long as T_i is running. For T_j the substitute transaction T^S_i anticipates the (future) write of T_i . T_i is protected by its substitute transaction so that either T_i does not have to validate again at all or at least the probability of positive validation is increased.

Substitute transaction

After a certain number of restarts of T_i the substitute transaction T^S_i is established such that $RS^S_i=RS_i$ and $WS^S_i=WS_i$. After establishing its substitute T_i is restarted. T^S_i exists as long as T_i is active. In contrary to other transactions the first action of a substitute transaction is to read its transaction number. From this moment, the status of T^S_i is that of a validating transaction and therefore any other validating transaction T_j must validate against T^S_i (for validation see below). The main point about substitute transactions is that they are not transactions which have written at a certain point in time, but which have to be considered in every validation as long as they exist.

Validation

A transaction T_j must apply the following test in its validation phase against an existing substitute transaction T^S_i:

$$WS_j \cap RS^S_i = \emptyset$$

To see this, note that T_j recognizes the substitute transaction, but T_i itself only writes after the termination of T_j (T^S_i is deleted at the end of T_i!). Of course, if T_i has written before the

validation of T_j, then T_j applies the standard validation test against T_i. Validation of a transaction T_j is performed in the way known from snapshot validation:

- ☞ During the read-phase of T_i, it validates against every terminating update transaction T_U with the usual test $WS_U \cap RS_i = \emptyset$.

- ☞ In the validation phase T_i has to validate against every transaction which has not yet ended, but which has read its transaction number before T_j (see chapter 3). This class of transactions includes all substitute transactions. Whereas validation against regular transactions is done with the usual test, T_j has to perform the test $WS_j \cap RS^S_i$ against substitute transactions.

Fixed or variable read- and write-sets of transactions

If the read and write sets of a transaction are the same for each restart, then the outlined scheme guarantees that a transaction with a substitute will terminate without further restart. In fact, the transaction needs not to perform a validation at its end, since its substitute gave it perfect shelter from interferences with other transactions.

If the read- and write-sets are time-dependent and thus may differ from one restart to the next, then, obviously, the substitute transaction does not give this perfect shelter, however it considerably increases the chance of successful termination. T_i must validate in this case, T^S_i makes it probable that validation will succeed. If T_i must nevertheless be restarted, a new substitute transaction TS_i' may be installed with $RS^S_i' = RS_i \cup RS^S_i$, $WS^S_i' = WS_i \cup WS^S_i$, where RS_i, WS_i are the read- and write-sets of the last execution of T_i. This again increases the probability for T_i to terminate. As can easily be seen, this scheme guarantees termination of T_i with the only exception that there is a continuous adding of database objects which must be included in the read set of T_i.

Maintenance of substitute transactions

Concurrency control must guarantee some form of fairness for the installation of substitute transactions, because otherwise a transaction might never succeed in getting its substitute, thus again being blocked permanently. The simplest solution to this problem is to allow only one substitute transaction at a time. Installation of the substitute transaction is no problem at all, since a substitute does not have a read phase. To guarantee fairness, concurrency control serves requests for substitute transactions on a first-come first-served basis. A more complicated scheme is to allow several substitutes to exist in parallel. In this case the substitute

transactions must not conflict one with each other. Let S be the set of existing substitute transactions. Then, a new substitute transaction T^S_i can only be installed, if it does not conflict with some T^S_j out of S:

$$RS_i \cap WS_j \cup RS_j \cap WS_i \cup WS_i \cap WS_j = \varnothing$$

To ensure fairness, again first-come first-served principle might be applied. Since each transaction which has a substitute transaction will terminate with certainty, each transaction waiting to get a substitute will get it in finite time. In a slightly modified form this argumentation also holds in the case of variable read sets.

The presented concept is a simple and smooth solution for the starvation problem in optimistic approaches. The solution only makes use of optimistic concepts, artificial concepts like locking need not to be introduced. The necessary extensions to the basic optimistic concurrency control are very easily implemented. Depending on the maintenance of the substitute transactions there is either no or only very small run-time overhead for the proposed solution. We have been informed about results from simulations, not yet published, which confirm the efficiency of the substitute concept.

8 Conclusion

The basic optimistic concurrency control scheme as proposed in [KuRo79] exhibits some serious shortcomings with respect to its validation technique and long transactions. In the first part of this paper design alternatives were presented which are superior to the original approach in several aspects. The main advantage of the proposed solution is that it distinguishes between serious and non-serious conflicts and only rolls back a transaction in case of a serious conflict. No additional overhead or other restrictions are introduced. On the contrary, overhead is reduced. In the next part of the paper a special validation scheme for read transactions was introduced. Under control of this proposal a read transaction is only backed up if a conflict is detected which definitely violates the serialization criterion. In the next part of the paper the integration of versions into the optimistic approach were discussed briefly. A concept was presented which allows read transactions to run without any consideration of concurrency control. Update transactions are not hampered by this support for readers. Of course, as in many version based approaches, there is additional overhead for the maintenance of versions, and it remains to be shown which circumstances these additional costs are worth paying. The last part of this paper presented a solution for the starvation problem. The

proposed algorithm is easy to integrate because it is only based on optimistic concepts, so that artificial concepts like timestamps or locking need not to be introduced.

Literature

[Agra83] Agraval, R.: Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance EvaluationP; Ph.D. thesis; University of Wisconsin, Madison; 1983

[AgCL85] Agraval, R., Carey, M., Livny M.: Models for Studying Concurrency Control Performance: Alternatives and Implications; Proc. ACM-Sigmod, International Conference on Management of Data; Austin, Texas; 1985

[AgCL87] Agraval, R., Carey, M., Livny M.: Concurrency Control Performance Modeling: Alternatives and Implications; ACM Transactions on Database Systems; Vol. 12, No. 4; Dec. 1987

[AuPS84] Augustin, R., Prädel, U., Scholten, H.: Performance Analysis of Concurrency Control Algorithm in Database Systems: a Survey (in German); Research-Report No. 174; Department of Computer Science, University of Dortmund, Germany; 1984

[Bada81] Badal, D.: Concurrency Control Overhead or Closer Look at Blocking vs. Non-Blocking Concurrency Control Mechanism; Proc. of the 5th Berkeley Workshop; 1981

[Bhar80] Bhargava, B.: An Optimistic Concurrency Control Algorithm and its Performance Evaluation against Locking Algorithms; Report of the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260 USA; 1980

[Bhar82] Bhargava, B.: Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking; Report of the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260 USA, 1982

[BoCa92] Bober, P.; Carey, M.: Multiversion Query Locking; Proc. 18th Int. Conf. on Very Large Databases (VLDB); Vancouver, Canada; Aug. 1992

[Care83] Carey, M.: Modeling and Evaluation of Database Concurrency Control Algorithms; Ph.D. thesis; University of California, Berkeley; 1983

[CFLN82] Chan, A.; Fox, S.; Lin, W.; Nori, A.; Ries, D.: The Integration of an Integrated Concurrency Control and Recovery Scheme; Proc. ACM-SIGMOD, International Conference on Management of Data; 1982

[DuBo82] DuBourdieu, D.: Implementation of Distributed Transactions; Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks; 1982

[FrRo85] Franaszek, P; Robinson, J. T.: Limitations of Concurrency in Transaction Processing; ACM Transactions on Database Systems; Vol. 10, No. 1; Mar. 1985

[GaRa92] Gafni, A.; Rao, B: A Time-Based Distributed Optimistic Recovery and Concurrency Control Mechanism; Proc. 8th Int. Conf. on Data Engineering; Tempe, Arizona; 1992

[HaCL90] Haritsa, J.; Carey, M.; Livny, M.: Dynamic Real-Time Optimistic Concurrency Control; Proc. 11th Real-Time Systems Symposium; Dec. 1992

[Härd84] Härder, T.: Observations on Optimistic Concurrency Control Schemes; Information Systems, Vol. 9, No. 2; 1984

[HSRT91] Huang, J.; Stankovic, J.; Ramamrithan, K.; Towsley, D.: Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes; Proc. 17th Int. Conf. on Very Large Databases (VLDB); Barcelona, Spain; 1991

[HuSt90] Huang, J.; Stankovic, J.: Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs Two-Phase Locking; Technical Report, COINS 90-121; University of Massachusetts; Nov. 1990

[JaSh92] Jagadish, H.; Shmueli, O.: A Proclamation-Based Model for Cooperating Transactions; Proc. 18th Int. Conf. on Very Large Databases (VLDB); Vancouver, Canada; Aug. 1992

[KeTe84] Kersten, M., Tebra, H.: Application of an Optimistic Concurrency Control Method; SOFT-WARE-Practice and experience, Vol. 14, Feb 1984

[KuRo79] Kung, H. T., Robinson, J. T.: On Optimistic Methods for Concurrency Control; Proc. 5th Int. Conf. on Very Large Databases (VLDB); Rio de Janeiro, Brazil; 1979

[Maie89] Maier, D.: Making Database Systems Fast enough for CAD Applications; in: Kim, W.; Lochovsky, F. (Editors): Object-Oriented Concepts, Databases, and Applications; Addison-Wesley Publishing Company; 1989

[MeNa82] Menasce, D., Nakanishi, T.: Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems; Information Systems, Vol. 7, No. 1, 1982

[PaKa84] Papadimitriou, C.; Kanellakis, P.: On Concurrency Control by Multiple Versions; ACM Transactions on Database Systems; Vol. 9, No. 1; March 1984

[PeRe83] Peinl, P., Reuter, A.: Empirical Comparison of Database Concurrency Control Schemes; Proc. of the 9th Int. Conf. on Very Large Data Bases (VLDB); Florence, Italy; 1983

[PrSU82] Prädel, U., Schlageter, G., Unland, R.: Ideas on Optimistic Concurrency Control I; Informatik-Berichte No. 26, University of Hagen, Germany, 1982

[Schl81] Schlageter, G.: Optimistic Methods for Concurrency Control in Distributed Databases; Proc. of the 7th Int. Conf. on Very Large Data Bases(VLDB); Cannes, France; 1981

[TaGS84] Tay, Y.; Goodman, N.; Suri, R.: Performance Evaluation of Locking in Databases: A Survey; Rechnical Report; TR-17-84, Harvard Aikon Lab.; Cambridge, Mass.; 1984

[Unla85] Unland, R.: Optimistic Concurrency Control and its Efficiency in Comparison to Locking (in German); Ph.D. thesis, University of Hagen, Germany; 1985

[UnPS83] Unland, R., Prädel, U., Schlageter, G.: Design Alternatives for Optimistic Concurrency Control Schemes; Proc. ICOD, Cambridge and Research-Report, University of Hagen, 1983

[UnPS86] Unland, R., Prädel, U., Schlageter, G.: Redesign of Optimistic Methods: Improving Performance and Applicability; Proc. 2nd Int. Conf. on Data Engineering, Los Angeles, 1986

[UnSc88] Unland, R., Schlageter, G.: Improved Optimistic Cuncurrency Control and its Use in Distributed Database Systems; Proc. 21th Hawaii Int. Conf. on System Sciences, Kona, Hawaii, 1988

[WaSL86] Wang, S., Singhal, M., Liu, M.: An Optimistic Concurrency Control Algorithm for Database Systems; Proc. Int. Computer Symposium Taiwan, Dec., 1986

Arbeitsberichte des Instituts für Wirtschaftsinformatik

Nr. 1 Bolte, Ch., Kurbel, K., Moazzami, M., Pietsch, W.: Erfahrungen bei der Entwicklung eines Informationssystems auf RDBMS- und 4GL-Basis; Februar 1991.

Nr. 2 Kurbel, K.: Das technologische Umfeld der Informationsverarbeitung - Ein subjektiver 'State of the Art'-Report über Hardware, Software und Paradigmen; März 1991.

Nr. 3 Kurbel, K.: CA-Techniken und CIM; Mai 1991.

Nr. 4 Nietsch, M., Nietsch, T., Rautenstrauch, C., Rinschede, M., Siedentopf, J.: Anforderungen mittelständischer Industriebetriebe an einen elektronischen Leitstand - Ergebnisse einer Untersuchung bei zwölf Untenehmen; Juli 1991.

Nr. 5 Becker, J., Prischmann, M.: Konnektionistische Modelle - Grundlagen und Konzepte; September 1991.

Nr. 6 Grob, H.L.: Ein produktivitätsorientierter Ansatz zur Evaluierung von Beratungserfolgen; September 1991.

Nr. 7 Becker, J.: CIM und Logistik; Oktober 1991.

Nr. 8 Burgholz, M., Kurbel, K., Nietsch, Th., Rautenstrauch, C.: Erfahrungen bei der Entwicklung und Portierung eines elektronischen Leitstands; Januar 1992.

Nr. 9 Becker, J., Prischmann, M.: Anwendung konnektionistischer Systeme; Februar 1992.

Nr. 10 Becker, J.: Computer Integrated Manufacturing aus Sicht der Betriebswirtschaftslehre und der Wirtschaftsinformatik; April 1992.

Nr. 11 Kurbel, K., Dornhoff, P.: A System for Case-Based Effort Estimation for Software-Development Projects; Juli 1992.

Nr. 12 Dornhoff, P.: Aufwandsplanung zur Unterstützung des Managements von Softwareentwicklungsprojekten; August 1992.

Nr. 13 Eicker, S., Schnieder, T.: Reengineering; August 1992.

Nr. 14 Erkelenz, F.: KVD2 - Ein integriertes wissensbasiertes Modul zur Bemessung von Krankenhausverweildauern - Problemstellung, Konzeption und Realisierung; Dezember 1992.

Nr. 15 Horster, B., Schneider, B., Siedentopf, J.: Kriterien zur Auswahl konnektionistischer Verfahren für betriebliche Probleme; März 1993.

Nr. 16 Jung, R.: Wirtschaftlichkeitsfaktoren beim integrationsorientierten Reengineering: Verteilungsarchitektur und Integrationsschritte aus ökonomischer Sicht; Juli 1993.

Nr. 17 Miller, C., Weiland, R.: Der Übergang von proprietären zu offenen Systemen aus Sicht der Transaktionskostentheorie; Juli 1993.

Nr. 18 Becker, J., Rosemann, M.: Design for Logistics - Ein Beispiel für die logistikgerechte Gestaltung des Computer Integrated Manufacturing; Juli 1993.

Nr. 19 Becker, J., Rosemann, M.: Informationswirtschaftliche Integrationsschwerpunkte innerhalb der logistischen Subsysteme - Ein Beitrag zu einem produktionsübergreifenden Verständnis von CIM; Juli 1993.

Nr. 20 Becker, J.: Neue Verfahren der entwurfs- und konstruktionsbegleitenden Kalkulation und ihre Grenzen in der praktischen Anwendung; Juli 1993.

Nr. 21 Becker, K., Prischmann, M.: VESKONN - Prototypische Umsetzung eines modularen Konzepts zur Konstruktionsunterstützung mit konnektionistischen Methoden; November 1993

Nr. 22 Schneider, B.: Neuronale Netze für betriebliche Anwendungen: Anwendungspotentiale und existierende Systeme; November 1993.

Nr. 23 Nietsch, T., Rautenstrauch, C., Rehfeldt, M., Rosemann, M., Turowski, K.: Ansätze für die Verbesserung von PPS-Systemen durch Fuzzy-Logik; Dezember 1993.

Nr. 24 Nietsch, M., Rinschede, M., Rautenstrauch, C.: Werkzeuggestützte Individualisierung des objektorientierten Leitstands ooL, Dezember 1993.

Nr. 25 Unland, R., Meckenstock, A., Zimmer, D.: Flexible Unterstützung kooperativer Entwurfsumgebungen durch einen Transaktions-Baukasten, Dezember 1993.

Nr. 26 Grob, H. L.: Computer Assisted Learning (CAL) durch Berechnungsexperimente, Januar 1994.

Nr. 27 Unland, R., Kirn, St.: Unterstützung Organisatorischer Prozesse durch CSCW, März 1994.

Nr. 28 Unland, R., Kirn, St.: Zur Verbundintelligenz integrierter Mensch-Computer-Teams: Ein Organisationstheoretischer Ansatz, März 1994.

Nr. 29 Unland, R., Kirn, St., Wanka, U.: Workflow Management mit Kooperativen Softwaresystemen: State for the Art und Problemabriß, März 1994.

Nr. 30 Unland, R.: Optimistic Concurrency Control Revisited, März 1994.

Nr. 31 Unland, R.: Sematics-Based Locking: From Isolation To Cooperation, März 1994.