Inform. Systems Vol. 9, No. 2, pp. 111–120, 1984
Printed in the U.S.A.

0306-4379/84 $3.00 + .00

OBSERVATIONS ON OPTIMISTIC CONCURRENCY CONTROL SCHEMES

THEO HÄRDER†
Department of Computer Sciences, University of Kaiserslautern, Postfach 3049, D-6750 Kaiserslautern, West Germany

(*Received* 18 *April* 1983; *in revised form* 6 *October* 1983)

Abstract—Optimistic concurrency control schemes allow uncontrolled access to shared data objects during transaction processing under the explicit assumption that read and write conflicts among transactions are rare events. Before a transaction commits, the DBMS has to validate that no conflict has occurred. Conflict resolution mainly relies on transaction abort.

Two different optimistic concurrency control schemes are introduced and compared to each other. The problems of implementing such schemes and their implications on DBMS processing is investigated in some detail. A number of general properties of optimistic concurrency control schemes is derived, and their advantages and drawbacks w.r.t. two-phase locking approaches are discussed.

INTRODUCTION

When transactions are accessing a database concurrently, a concurrency control (CC) scheme has to prevent conflicts among them such that their serializability can be guaranteed. A system of concurrent transactions is said to be serializable or has the property of serial equivalence if there exists at least one serial schedule of execution leading to the same results for every transaction and to the same final state of the database.

Conventional CC schemes use two-phase locking protocols acquiring dynamically locks for the objects to be accessed. Since the implementation of predicate locks appears to be impractical, usually physical locks are used. Special care has to be taken in such implementations to prevent errors based on the non-existence of objects, e.g. phantoms[1]. Approaches with appropriately implemented two-phase locking protocols guarantee a single user system view except for deadlocks.

It is claimed[2] that locking approaches have the following inherent disadvantages:

—Lock maintenance and deadlock detection represent a substantial overhead, e.g. 10% of the total execution time in System R[3].

—There are no general purpose deadlock-free locking protocols that always provide a high degree of concurrency.

—Concurrency is significantly lowered whenever it is necessary to leave some hot spot data object (congested node) locked while waiting for a secondary memory access.

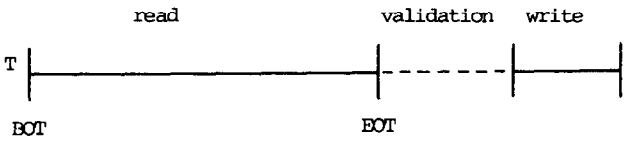
—Because of the presence of failures a strict two-phase locking protocol has to be applied to prevent backout of data which may have been accessed by completed transactions before the failing transaction aborts, that is, locks have to be kept until EOT.

—Locking may be necessary at all only in the worst case, that is, in most cases a locking approach introduces too strong preventive measures.

PRINCIPLES OF OPTIMISTIC CC

Optimistic CC schemes[2, 4, 5] are designed to get rid of the locking overhead. They are optimistic in the sense that they take into account the explicit assumption that conflicts among transactions are rare events. Thus, they rely for efficiency on the hope that conflicts will not occur. Since dynamically requested locks are not applied, such schemes are deadlock-free. The burden of CC is deferred until EOT when some checking for potential conflicts has to take place. If a conceivable conflict is detected, a "pessimistic" view has to be taken: this conceivable conflict is resolved by aborting the transaction. Hence, these schemes rely on transaction backout as a control mechanism.

The basic idea of an optimistic CC scheme is as follows: The execution of a transaction consists of three phases—read, validation, and write.



†This work was performed while the author was visiting IBM Research Laboratory, San Jose, CA 95193, U.S.A.

The end of the read phase corresponds to the signalling of EOT to the DBMS, that is, the transaction gives up its right of unilateral abort.

Read phase

For each transaction a so-called transaction buffer (implemented as an intention list) is maintained under control of the DBMS. This buffer supports the abortion of a partially executed transaction during its read phase. Moreover, a completely executed, but conflicting transaction can be backed out during its validation phase without causing any effect to the global database state. During the read phase a read access is first directed to the transaction's buffer. If the object is not found in the buffer, the (global state of the) database has to be accessed, that is, the system buffer or the data stored on disk. Read-only objects may or may not be cached in the transaction's buffer for later use. Modified objects, however, are stored in the transaction's buffer. Repeated modifications (insertions, deletions and updates) on the same objects within a transaction are made on its local copies.

Note, whenever the modifications during the read phase are only applied to the abstract view of the objects visible at the DB programming interface, the underlying access path data are not affected at that time. This strategy preserves small granules of conflict (records), but has far-reaching consequences on DBMS processing (see discussion below). Using copies of the original pages for the transaction's modifications, all corresponding access path data can be changed immediately. This page-level modification greatly facilitates normal DBMS processing. However, the granule of conflict becomes much larger having potentially strong influences on the outcome of the optimistic CC. (Direct modification of the global database state during the "read phase" without having acquired the appropriate locks seems to lead to timestamping schemes, as another class of CC[6, 7].

Validation phase

After signalling EOT (prepared to COMMIT), the DBMS has to check whether or not the transaction intending to commit was in conflict with any of the transactions operating in parallel. Since no locks are held, the objects read by the transaction might have been modified meanwhile by concurrent transactions. If so, some conflict resolution policy has to be applied. If no conflict is detected, the transaction is prepared to commit.

Write phase

A reader transaction is automatically committed after successfully completing the validation phase. A writer transaction has to force sufficient log data to a safe place, consisting at least of the REDO information for the transaction's modifications, e.g. for archive recovery purposes in case of a media failure. This action completes phase 1 of a two-phase COMMIT protocol.

Then the modifications can be propagated to the global database state. "Write" does not necessarily mean "output to disk". It only requires to make the transaction's modifications generally visible, e.g. in the system buffer. The question whether or not they should be immediately forced to disk or replaced by normal buffer management is of no importance to the principle of optimistic CC. The committed data has to be secured in phase 2 and eventually later on against system failures[8, 9].

On the first sight, optimistic CC schemes appear to be very appealing and seductive, since they seem to promise getting rid of waiting times, deadlock problems, and the management of lock control data inherent in conventional locking schemes. The proposed solutions—demonstrated at a rather high level of abstraction—seem to be reasonable and elegant. In particular, many implementation problems inherent in these schemes disappear at the abstract level chosen for the discussion or are hidden when only dealing with read and update operations on simple objects. Insertion and deletion is assumed to be covered by these operations, e.g. all potential objects are somehow represented in the database—insertion of an object means its transition from a special null state to some meaningful state and deletion of an object vice versa. Due to these assumptions which hardly seem to be achievable in practical applications these schemes get rid of the messy phantom-related problems which are discussed below.

VALIDATION OF SERIAL EQUIVALENCE

Since no restrictions are applied to read accesses during the read phase, it must be assured during the validation phase of a transaction that all its read accesses were directed to the same database state. This, in turn, implies that the result of the transaction is equivalent to its execution in some serial schedule.

In order to validate the serial equivalence criterion, a unique transaction number T_i is assigned to a transaction at the end of its read phase (in fact, the actual assignment can be postponed until successful validation). Each time a transaction number is assigned, the transaction number count (TNC) is incremented by 1. If T_i finishes its read phase before T_j (i.e. T_i is validated before T_j), then $T_i < T_j$ holds.

For each T_j and for all T_i with $T_i < T_j$, there must exist some serially equivalent schedule with T_i executed before T_j. In order to be serializable two transactions T_i and T_j must observe the following two rules:

Rule 1: *No read dependency*

Rule 11: T_i does not read data modified by a concurrent transaction T_j.

Rule 12: T_j does not read data modified by a concurrent transaction T_i.

Rule 2: *No overwriting*

T_i does not overwrite data which has been written by a concurrent transaction T_j and vice versa.

There are two basic ways to guarantee rules 1 and 2:

(a) no time overlap between T_i and T_j;

(b) no object set overlap (data overlap) between T_i and T_j.

(a) and (b) can be applied separately to read sets (RS) and write sets (WS) to construct more sophisticated protocols.

This leads to the following set of alternatives for optimistic CC:

1. *No time overlap at all*
   Serial execution of T_i and T_j which trivially guarantees rules 1 and 2.

2. *No time overlap of write phases*
   2.1 T_j does not read data being modified by T_i (no object set overlap). This guarantees rule 12.
   2.2 T_i completes write phase before T_j starts write phase (no time oberlap; T_j cannot affect the read phase of T_i) This guarantees rules 11 and 12.

3. *No object set overlap of write sets*
   3.1 T_j does not read data being modified by T_i (no object set overlap). This guarantees rule 12.
   3.2 T_i does not read data being modified by T_j (no time overlap between read phase of T_i and write phase of T_j; T_j cannot affect the read phase of T_i). This guarantees rule 11.
   3.3 Write sets of T_i and T_j are disjunct (no object set overlap allowing concurrent execution of write phases). This guarantees rule 2.

4. *No object set overlap of read sets and write sets*
   Totally independent concurrent execution which trivially guarantees rules 1 and 2.

Alternatives 1–3 are a reformulation of the serializability conditions given in [2]. Alternative 4 is a trivial addition (for symmetry reasons). It is a little bit stronger than alternative 3, but does not allow for greater concurrency. It should be emphasized that these alternatives represent sufficient, but not necessary, conditions for serializability[5].

Alternative 1 is obvious; no care has to be taken when transactions do not overlap in time. The chosen transaction numbering scheme controls this condition.

For concurrently executed transactions, alternative 2 provides the basic approach for optimistic CC. To prevent time overlap of write phases, only one committing transaction can be accepted at a time. If COMMIT processing can be performed efficiently, enforcement of this condition seems to be sufficient.

Alternative 3 (and 4) allows n committing transactions at a time provided that the corresponding conditions are satisfied. At the expense of higher complexity, some parallelism might be gained during COMMIT processing. In[2] an algorithm for parallel validation is given intending to shrink the time interval needed for exclusive DB control on behalf of the committing transaction. Additional problems are introduced (a committing transaction invalidates a transaction, even though the former transaction is itself invalidated). These algorithms tend to become very complex even although their underlying model of data and operations is very simple, i.e. they do not have to maintain hidden access path data and system control structures. Therefore, the expected gain of these algorithms might be compensated by additional complexity and overhead in real DB implementations.

We concentrate on optimistic CC schemes which are based on alternative 2. Two different approaches are discussed in the following.

BACKWARD ORIENTED OPTIMISTIC CC

Backward oriented optimistic CC (BOCC) checks during the validation phase of T_j whether its read set $RS(T_j)$ intersects with any of the write sets $WS(T_i)$ of all concurrently executed transactions T_i having finished their read phases before T_j. Since "blind" modifications are not very likely, each transaction has to be validated in practice.

Let T_{start} be the highest transaction number assigned to some transaction when T_j starts, and T_{finish} the highest transaction number when T_j enters its validation phase. Then, essentially the following procedure in T_j's validation phase will decide T_j's destiny.

```
VALID:= TRUE;
FOR Ti:= Tstart+1 TO Tfinish DO
  IF RS(Tj)∩WS(Ti)¬ = ∅ THEN
     VALID:= FALSE;
IF VALID THEN COMMIT
         ELSE ABORT;
```

The scenario shown in Fig. 1 illustrates the set of transactions to be checked.

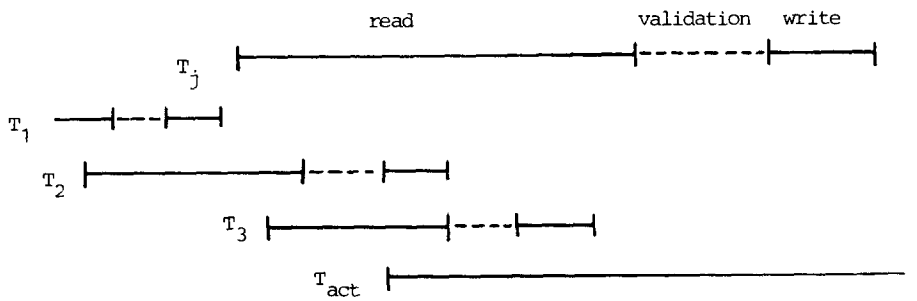


Fig. 1. Validation scenario for transaction T_j (BOCC).

BOCC requires all finishing transactions—readers or writers—to be tested whether or not they satisfy the serializability criterion as long as they have read some data (RS = {0}). Of course, for "blind" write only transactions there is nothing to check. It allows no flexibility in the COMMIT or ABORT decision to be made.

In the given scenario, RS(T_j) has to be checked against WS(T_2) and WS(T_3). Since T_2 and T_3 have already committed, the only conflict resolution strategy is to abort if there is an intersection.

The given procedure allows exactly one validating and committing transaction at a time. It has to be executed in a "critical section" w.r.t. other transactions intending to commit. Within this critical section, all other transactions can proceed with their read phases. If the serialization of validation and write phases is a problem, part of the validation can be executed in parallel. Let us assume that T_j has been validated against T_{start} to T_{finish} outside of a critical section. Since new transactions ($T_{finish+1}$ to $T_{finish'}$) could have committed in the meantime, they have to be checked additionally. If this is done in parallel, again some transaction could have committed requiring further checking. In order to prevent being passed forever, eventually a critical section has to be entered for the final validation step and for the write phase.

Note, the read sets of committed transactions are of no interest any more. But all write sets of overlapping transactions (which are static sets) have to be kept until their last concurrent transaction has finished. In case of long transactions having a large number of concurrent writers this requirement might be a strong handicap.

FORWARD ORIENTED OPTIMISTIC CC

Forward oriented optimistic CC (FOCC) checks during the validation phase of T_j whether its write set WS(T_j) intersects with any of the read sets RS(T_i) of all transactions T_i having not yet finished their read phases. This strategy assures that read sets are always clean. Write sets are only propagated if they do not conflict with the current read sets of all other active transactions.

Let the active transactions have the numbers T_{act1} until T_{actn}. Then, T_j is validated as follows:

```
VALID:= TRUE;
FOR Ti:= Tact1 TO Tactn DO
  IF WS(Tj)∩RS(Ti)¬ =∅ THEN
     VALID:= FALSE;
IF VALID THEN COMMIT
          ELSE RESOLVE CONFLICT;
```

The scenario shown in Fig. 2 illustrates the set of transactions to be checked.

FOCC places the burden of CC exclusively to writers. It requires only finishing writer transactions to be tested whether or not they satisfy the serializability criterion. Hence, readers will be automatically committed, once they reach their EOT.

Since the transactions to be checked during validation have not yet committed, this approach offers a great deal of flexibility in handling a detected conflict. In the given scenario, WS(T_j) has to be checked against RS($T_{act\ 1}$) and RS($T_{act\ 2}$). When a conflict is realized, the following resolution strategies are conceivable:

1. *Defer due to conflicting readers*

When the set of conflicting transactions are readers, the validating transaction T_j may be deferred; validation has to be retried later. The conflicts are resolved as soon as all conflicting readers have finished. Of course, this strategy embodies an optimistic view even during validation. It is true that the current transactions will finally terminate (assume they are correct), but new readers might cause further conflicting accesses in the meantime. As mentioned earlier, the risk of indefinite delay is present in all CC schemes based only on transaction abort and has to be cured sooner or later by some drastic serialization protocol. Provided that conflicts are rare events, this strategy expands nicely the idea of optimistic CC.

2. *Defer due to conflicting writers*

Even with some writers in the set of conflicting transactions, the defer strategy may provide a non-hurting conflict resolution, when the "future" write sets of such conflicting transactions are disjoint with the write set of T_j. Assume T_j has modified A, B to

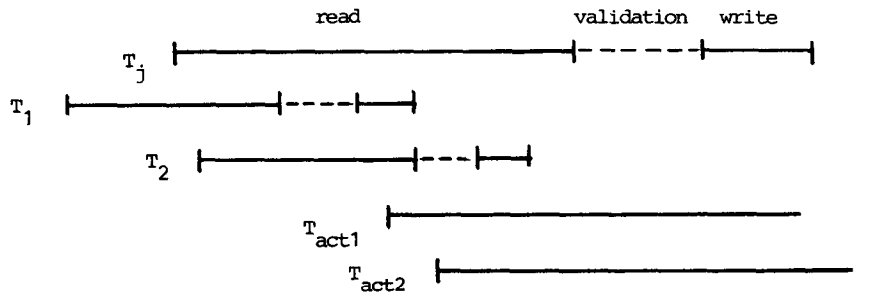


Fig. 2. Validation scenario for T_j (FOCC).

A', B', and that the writer T_{act1} has currently read A, B and C, while preparing C'. Deferring T_j until T_{act1} has committed gurarantees the serialization order $T_{act1} < T_j$. This strategy, however, represents an even more optimistic view of the world with all the implications of the first strategy.

3. *Kill and commit*

Since all conflicting transactions have not yet committed, the obstacles can be removed deliberately. Assume T_j is a long writer, and T_{act1} has just started its execution. In such cases, a killer scheme might be advantageous.

A weaker interaction would consist of invalidating the conflicting transactions' read sets. An invalidated transaction might survive, if it decreases its level of consistency, e.g. does not rely on repeatable reads.

4. *Abort*

The validating transaction T_j is immediately aborted.

With the choice of a particular strategy or a mix of them, a priority scheme for readers or writers can be supported.

A critical point of FOCC is the checking of dynamic read sets which has to be performed whenever a writer commits. The simplest solution is to quiesce system activity (no concurrent progress in read phases) and to check all active lists in a critical section. Since this may not be tolerable for performance reasons, some mechanism permitting parallel activity in read phases has to be invented (see below).

Note, FOCC-strategies seem to have many common properties with locking protocols granting only S-locks during the read phase. A writer must try to convert its S-locks for modified objects into X-locks at EOT. If no sharing occurs, the corresponding X-lock is immediately granted. If several S-locks are held on an object, the conversion must be deferred. An important difference, however, is introduced by the version control. The lock protocol automatically precludes different versions of the same object and makes sure that the same version is repeatedly granted to the requesting transaction, whereas in FOCC-strategies version control has to be achieved by some special mechanisms (e.g. token schemes).

In comparing the BOCC- and the FOCC-strategies, the following observations seem to be important:

—The write set is often a (small) subset of the read set; if not, then the disjoint fraction is caused by new insertions which hardly lead to conflicts (at the record level).

—BOCC must validate a potentially large read set against a potentially large number of old (static) write sets. The read set and the number of write sets increase with the duration of the validating transaction.

—FOCC validates no read sets. It has to check a (small) write set against a limited number of concurrent (dynamic) read sets. The sizes and the number of the read sets is not dependent on the length of the validating transaction.

—Validation in FOCC-strategies is more difficult (allowing concurrent activity in read phases is more costly), because dynamic sets have to be examined. Since only writers are subject to validation, checking is much less frequent compared to BOCC-strategies.

Since the read set of a transaction has to be controlled anyway during its lifetime, less overhead should be expected for FOCC-strategies. It is also safe to say that FOCC-strategies offer more degrees of freedom in handling and optimizing conflict resolution.

IMPLEMENTATION CONSIDERATIONS

Optimistic CC seems to be conceivable with objects of different levels of abstraction. Obviously, the following two are prime candidates:

—high level objects of the data model, e.g. records;

—objects at the storage structure level, e.g. pages.

Record-level CC requires a list of modified records to be kept in the transaction buffer until the write phase is executed. The propagation of the modifications to their home pages including the corresponding access path structures has necessarily to take place during the write phase. To obtain a realistic implementation for such optimistic CC schemes with acceptable COMMIT processing times[9], it must be assumed that a NOSTEAL-property for a transaction's modified objects can be maintained, that is, that the corresponding pages can be kept in the DB system buffer until its write phase is executed. Without this assumption pages to be modified have to be reread from disk in the critical section whenever they have been replaced from the system buffer during a transaction's lifetime. A FORCE-policy (writing all modified pages to disk at EOT) should be avoided, because all pages necessarily have to be written during the write phase. Hence, a NOSTEAL/NOFORCE-strategy seems to be indispensable for satisfactory COMMIT processing times.

Page-level CC allows the preparation of all modifications including access path structures in copies of their corresponding home pages during the read phase. Hence, very short write phases can be achieved. With a NOSTEAL/NOFORCE-strategy the write phase only consists of making the modified pages generally available in the system buffer (and of saving sufficient log information). Even a STEAL/FORCE-stragtegy which may obtain acceptable COMMIT processing times seems to be achievable with an ATOMIC propagation scheme[9], e.g. a transaction oriented shadow page mechanism (TOSP) which has to be tailored to maintain a transaction's modified pages on disk. Such a mechanism would imply

—to tentatively write the modified pages from the transaction buffer to disk during to read phase;

—to apply a logging technique to the modified

entries of the page tables in order to avoid synchronous I/O of page table blocks;

—to propagate the set of pages atomically after successful validation during the write phase.

In either case, COMMIT processing must be a primary concern for the implementation of an optimistic CC scheme for performance reasons. Note, in a virtual OS environment page faults can expand its duration even with NOSTEAL- or NOFORCE-assumptions.

In order to speed up the validation process appropriate data structures have to be designed for main memory use. For each transaction, two separate data structures should be maintained for its associated RS and WS. These two structures could be merged into one when special read and write flags are used for its entries. Each entry—called token—describes an object accessed with its name (e.g. TID or page—#). Since these data structures have to be compared efficiently with each other, bit list representations would be an appropriate choice; because of their size they are impractical in most applications. Compressed bit lists, however, require encoding and decoding operations for their modification and decoding operations for their comparison. Some sort order seems to be mandatory when tables or linked lists are used. Tables would require high overhead for the sorted insertion of entries. Linked lists imply linear search for look-up operations. Hence, the actual implementation has to balance these conflicting requirements.

BOCC schemes must keep information concerning the WS of all committed transactions which have some time overlap with running transactions. Maintaining a global list/table of all WS of the corresponding transactions is difficult because of garbage collection. Such an approach would avoid the need of keeping duplicate tokens for an object and of checking them several times. Therefore, a global table pointing to the lists/tables of tokens is proposed according to the scheme as shown in Fig. 3.

The global table could be organized as a circular table. The current active range (CAR) of WS should be implemented as a sliding range, since WS can be dropped as soon as their related transactions don't have any time overlap with running transactions. LBP indicates the lower bound of CAR; for the upper bound, a pointer derived from TNC can be used. The entries T_{start} and T_{finish} indicate the subrange in the global table which has to be checked against RS (T_{act}). After successful validation of T_{act}, the next available transaction—# is derived from TNC and assigned to T_{act}; WS(T_{act}) is attached at the end of CAR to the global table. LBP is given by the lowest T_{start} value stored in the RS-structures of all running transactions. When a transaction terminates (commit or abort), its T_{start} value is compared with LBP. If they are equal (mod table size), LBP is advanced to the minimum of the T_{start} values of the remaining active transactions; the WS being out of CAR are subject to garbage collection.

A conflict occurs when the circular table overflows (TNC tries to overwrite LBP) due to some long running transaction. Aborting the corresponding transaction resolves this conflict. Nevertheless, since a large number of WS have to be maintained for long times, serious storage management problems may arise.

FOCC strategies allow a less sophisticated implementation, since they do not rely on the checking of old WS. It is sufficient to keep information for the RS and WS of all active transactions as described. Validation is straightforward as long as it is done in a system-wide critical section. When parallel read phase actions are permitted, checking is more intricate because the RS are dynamic. One solution could consist of collecting the (new) concurrent reads in a special list in addition to their maintenance in their RS. This special list would have to be checked in a second step in a critical section.

GENERAL PROPERTIES OF OPTIMISTIC CC SCHEMES

Without going into further detail the following general properties of optimistic CC schemes based on record- or page-level are stated:

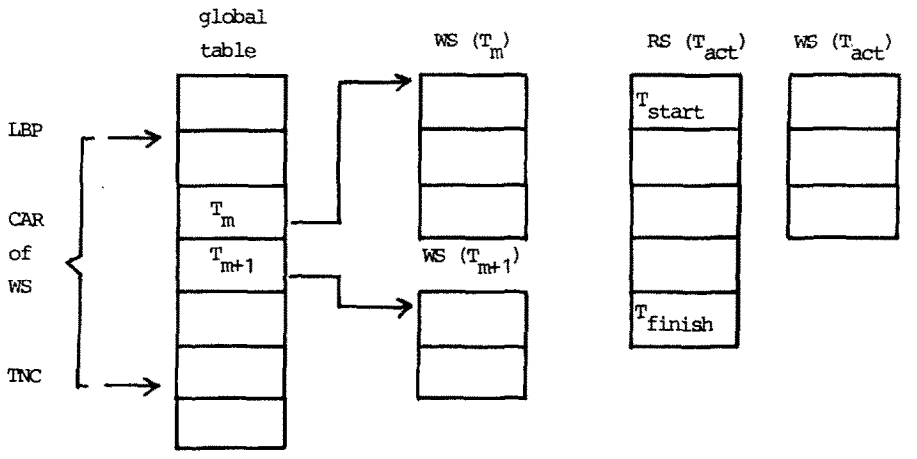


Fig. 3. Maintenance of token lists/tables in BOCC schemes.

Simple in-transaction backout

CC schemes with an intention list implementation (transaction buffer) are ideally suited for transaction abort due to user/data error, violation of restrictions, timeout, etc. Of course, locking schemes used together with intention lists provide the same kind of advantage.

Higher rate of transaction abort

They also allow a very efficient backout when a conflict is detected during validation. Such conflicts are resolved in locking schemes either by transaction wait or by deadlock detection and resolution.

It is assumed that transaction wait normally diminishes the risk of conflict. Hence, a higher fraction of transactions has to be expected to be aborted by using optimistic CC schemes. This is confirmed by empirical results[10] which compare locking protocols with optimistic CC schemes by using simulation experiments based on reference strings of various real-life DB applications. This study reveals clearly that validation conflicts in optimistic CC schemes cause transactions to be aborted to a much higher percentage than do deadlocks in the locking oriented schemes (with a parallelism of 32 transactions 36% compared to 10%).

Fair scheduling

When aborted transactions are reexecuted, there is a certain probability that they are backed out again. In order to obtain a sufficiently fair scheduling an appropriate conflict resolution strategy should be applied. Such requirements rule out optimistic CCschemes in their backward oriented version because their only choice is rolling back the unsuccessfully validating transaction.

FOCC schemes, however, exhibit greater flexibility. It is reported in [10] that the pure kill strategy achieved the best throughput. This is due to the fact that no validation ever fails. To observe the fair scheduling requirements, transactions already aborted several times must be favored in a conflict situation. Therefore, the pure kill scheme has to be modified appropriately.

Need of serialization

The probability of transaction abort seems to be strongly dependent on the length of the transaction (number of objects touched). Especially long transactions with overlapping read/write sets tend to produce permanent validation conflicts such that they all have to be reexecuted over and over again. To limit the thrashing situations and to solve the livelock problem, these critical transactions must be enabled to commit with a few restarts in the worst case. Strict serialization has been proposed to be enforced by some exclusive locking protocol in [2]. A similar approach using a dynamic load balancing algorithm with lock-out of transactions was chosen in [10].

Storage overhead

Due to the storage overhead of tokens and records/pages in virtual memory, optimistic CC restricts itself to rather short writer transactions. As discussed earlier, STEAL for pages is only practical in combination with complex mapping schemes, e.g. TOSP. Overflow of token lists to DASD data sets, however, has to be avoided for performance reasons. A load transaction would definitely cause problems. On the other hand, optimistic CC shows its special strength in a scenario with one long writer and many short readers proceeding without blocking and waiting times. By the way, such cases may not be typical for database processing environments.

Even long reader transactions accessing a key range or an entire file would consume a considerable fraction of control space (e.g. for tokens). Such transactions may be quite common in DB environments.

When locking schemes are used, there is no need of private data buffers. Of course, a load transaction would also cause storage overflow problems when executed in a NOSTEAL-environment. Storage overhead of lock representation can be avoided with locking schemes having some notion of hierarchy.

Control of phantom problems

In order to guarantee a consistent view of data in a multi-user DBMS, all user actions must be equivalent to the same sequence of actions in a serialized system. A well chosen implementation of an optimistic CC scheme (e.g. based on tokens of all physical objects touched) provides some level of consistency, e.g. it prevents error types like lost update, inconsistent analysis, dependency on uncommitted updates, etc. However, it does not provide full consistency, since access of sequence of records (including missing records) are not reproducible.

Phantom problems caused by the non-existence of objects or the interference of concurrently created objects having some semantic relationships to existing objects are not prohibited without special preventive measures. For example, token schemes based on records do not handle the case where records are missing during read operations, later inserted by another transaction, and then re-read by the original transaction. Since the records were originally missing, there are no tokens to validate, and therefore no way to catch that a logical conflict has occurred. This problem can be alleviated to some degree, if page-level tokens are chosen and the insertion of a "missing" record can be detected via the access path structures which it belongs to.

In locking schemes, phantom problems can be prevented by using hierarchical locks, e.g. at the relation or segment level. On the other hand, the granules of locking should be chosen as small as possible. By using some tricks, preventive measures for missing records can be introduced at the record (entry) level. They include, for example, locking of key ranges in index structures, locking of predecessor

and successor in link structures, locking of "end of file" indicators, locking of TIDs to exclude their reusage, etc. When no particular access path is available to determine the location of the "non-existing" record or the sequence of records to be read, a hierarchical locking scheme helps to prevent phantoms, e.g. for a relation scan, the entire relation is locked[11].

Of course, such tricks could also be introduced into an optimistic CC scheme. For example, a successor or predecessor token of a missing record has to be inserted into the read set of the requesting transaction. A writer has to keep in addition to the token for the newly inserted record the token of the successor/predecessor in its write set. In order to validate these tokens, special interpretation rules have to be observed.

Such enhancements at the record/page level do not seem to provide a general solution to the phantom problem. Assume a relation scan which keeps tokens of all pages accessed. When the underlying segment allows dynamic growth, there might still be some problems. Therefore, an extension of the optimistic CC scheme seems to be necessary. For example, such phantom problems can be handled by a hierarchical token scheme analogous to the locking approach. It is debatable, however, whether this extension is in the spirit of the optimistic view because of the large granule of conflict, i.e. a relation-level token is in conflict with all record tokens of the resp. relation. A writer transaction modifying a single record causes—probably fictitious—conflicts with *all* concurrent readers on behalf of which a relation-level token is used. Since there is a high probability of transaction abort it would be definitely better to delay these transactions by use of locks.

Time-consuming FORCE schemes

If the modified objects of a transaction are forced to the materialized database on disk at EOT to avoid partial REDO in case of a system crash, with record-level CC all the page I/O has to be done in a synchronous way at EOT causing potentially long delays. Page-level CC allows greater flexibility when combined with ATOMIC propagation schemes.

Deferred checking of consistency constraints

With record-level CC, certain integrity constraints cannot be checked during the read phase, e.g. simple unique key conditions or complex conditions like "average salary in department x less than y". They have to be postponed to the COMMIT phase. (Even when they would not be satisfied at the actual modification time, a truly optimistic view should rely on the hope that things change in favor of the checking transaction.) Hence, immediate checking of integrity conditions would sometimes save unnecessary work. But this requires page-level CC.

Deferred modification of access path data

Using record-level CC in a DB environment with a rich variety of access path structures, a potentially high overhead is created during EOT—even with NOSTEAL-policies and NOFORCE-strategies due to hidden costs caused by lower level objects. Assume the insertion of a new record which has to be propagated in the critical section. Assume further that the corresponding record type has n (say 17) index structures implemented as B^*-trees. Since these access path structures were not touched during the read phase, they are not found in the system buffer. Hence, $n*h$ pages (h = avg. height of a B^*-tree) have to be fetched in the worst case in order to modify the related index structures of the record type. Neglecting split operations, updates of free placement information and tables used for indirect addressing (DBTT = database key translation tables), etc. a substantial (overhead) time span is consumed. Note, these costs apply for the ideal combination scheme NOSTEAL/NOFORCE. In addition, a few I/O's have to be taken into account for writing REDO-information to the log. When a FORCE-strategy is used, massive synchronous I/O operations have to be expected even in the scenario with only one record insertion triggering a number of index modifications.

The deletion or update of an object read by some access path is also a lengthy operation when some additional index structures are involved. Hence, all modification operations may require a large number of I/O's in the critical section.

Complexity of query processing

The deferred index modification points to another problem which can cause all sorts of strange effects and complexities with record-level CC. Assume a transaction having inserted a record (in its local buffer) directs a query (How many . . .) to the DBMS which uses an appropriate index structure for its evaluation. Assume further that the newly inserted record belongs to the qualification set of the query. Such statistical queries can often be answered by looking only to an index entry. When the "optimized" COUNT-function does not include the local insertions of the issuing transaction, the wrong answer is returned. In order to correct such pecularities, a substantial effort has to be made in implementing query evaluation (complicating selection, join, view construction, etc. considerably).

The same problems arise with queries of transactions having deleted or updated records in their local buffers. Therefore, special code has to be executed for all kinds of queries, because these records are still present in the index structures used by the query evaluation. Hence, with record-level CC all modification operations will cause an increased complexity of the query evaluation process.

Page-level CC avoids these complexities when some indirect page addressing scheme is used.

Use of record-level CC

Because of the arguments concerning modification and query evaluation and the problems involved with

access path structures, record-level CC probably works only well in applications with simple file structures and a few simple transaction types where main memory operations can be guaranteed at COMMIT processing. In fact, a particular record-level CC scheme based on predicate testing is already implemented in IMS Fast Path for main storage database applications[12].

Page-level CC seems to be the only choice for optimistic CC in a complex DB environment.

Drawbacks of page-level CC

Using page-level CC, each record modification is prepared in the local buffer using a copy of its home page. Also, all related index pages are fetched during the read phase and updated in advance. Each page access has to be accompanied with a look aside to the transaction's buffer. This approach may consume large portions of local storage space or need to support some overflow scheme. Its main disadvantage, however, seems to be the increased granule of conflict (pages), which may considerably worsen the ratio of transactions to be aborted (even if no real conflict at the level of the data model is present). The larger the granule of conflict, the more likely are fictitious conflicts. The situation becomes particularly bad if it coincides with high traffic data elements.

For example, in many DBMS the available storage space is attempted to be occupied consecutively, that is, insertions of different transactions are allocated in the same page. While this strategy does not cause any trouble with record-level CC, it may produce fictitious conflicts with page-level CC. In data entry applications, the "current" hot spot page would provide some serious performance problems. (Arbitrary distribution of newly inserted records may lead to lower space utilization and may contradict value- or time-based clustering.)

Hence, the serialization of locking approaches on hot spot data pages due to an unsuitable locking granule may be turned into thrashing transaction aborts in such an optimistic CC scheme.

COMPARISON WITH LOCKING APPROACHES

Locking schemes guarantee one consistent image of the database at every point in time. For this purpose, they may sacrifice some degree of potential parallelism. However, they provide the facility of selecting an appropriate level of control to alleviate contention-related problems.

Optimistic CC schemes allow the uncontrolled creation of private data copies during the transactions' execution for the sake of enhanced concurrency. Their essential problem consists of merging these copies during COMMIT processing thereby regaining a transaction-consistent database image. As previously discussed, a lot of processing difficulties do arise, when these copies do not match with the units of transfer (pages). Hence, in order to be practically feasible for DBMS use, optimistic CC seems to be linked to the page level. As shown in various system implementations[11, 13], the locking approach has no restrictions of that kind. It can be chosen for the record or even for the field level.

In addition, particular assumptions—NOSTEAL/NOFORCE or ATOMIC propagation to the materialized DB—have to be introduced to minimize COMMIT processing or to reduce main storage use. Locking does not necessarily require such an environment for efficient transaction processing.

Note, concurrency control protocols in a DBMS are not designed for one special purpose only. Nevertheless, it is often argued that an optimistic CC approach should be chosen in applications where conflicts are unlikely. Since locking also behaves quite well in such a particular environment (no wait or deadlock conflicts), there seems to be little reason to introduce a specialized control mechanism. For a design decision concerning concurrency control, the following properties and requirements should be carefully regarded:

(1) Hot spot data need controlled serialization.

(2) If waiting situations and deadlocks are unlikely, locking is as good as optimistic CC.

(3) Each system needs some control hierarchy in order to provide efficient read and write operations on large data sets. For example, operations like loading a file, deleting a file, searching a file sequentially can be supported appropriately by some hierarchical locking scheme.

(4) Locking seems to be better suited to handle non-existence problems of records.

Acknowledgements—I would like to thank I. L. Traiger for many fruitful discussions and hints concerning the subject of this paper. His support during the preparation is greatly appreciated. I also thank W. Effelsberg, C. Mohan, P. Peinl and A. Reuter for their comments and suggestions. The comments of the referees are gratefully acknowledged.

REFERENCES

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger: The notions of consistency and predicate locks in a database system. *CACM* **19**(11), 624–633 (1976).

[2] H. T. Kung and J. T. Robinson: On optimistic methods for concurrency control. *ACM TODS* **6**(2), 231–226 (1981).

[3] J. N. Gray: Notes on data base operating systems. In *Lecture Notes in Computer Science*, Vol. 60. Springer-Verlag, Berlin (1978).

[4] G. Schlageter: Optimistic methods for concurrency control in distributed database systems. *Proc. 7th Int. Conf. on VLDB*, Cannes, France, 1981.

[5] S. Ceri and S. Owicki: On the use of optimistic methods for concurrency control in distributed database. *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 117–129. Asilomar, Feb. 1982.

[6] R. H. Thomas: A majority consensus approach to concurrency control. *ACM TODS* **4**(2), 180–209 (1979).

[7] P. A. Bernstein, D. W. Shipman and B. Rothnie: Concurrency control in a system for distributed database (SDD-1). *ACM TODS* **5**(1), 18–51 (1980).

[8] J. N. Gray *et al.*: The recovery manager of the system R database manager. *ACM Computing Surveys* **13**(2), 223–242 (1981).

[9] T. Härder and A. Reuter: Principles of transaction oriented database recovery—a taxonomy. *Research Report*. FB Informatik, Universitaet Kaiserslautern, April 1982.

[10] P. Peinl and A. Reuter: Empirical comparison of database concurrency control schemes. *Proc. 9th Int. Conf. on VLDB*, Florence, Italy, 97–108, 1983.

[11] M. M. Astrahan *et al.*: System R: a relational approach to database management. *ACM TODS* **1**(2), 97–137 (1976).

[12] IBM Corp. IMS/VS Version 1, Fast path feature. General Information Manual, GH20-9062-2. IBM, White Plains, N.Y., April 1978.

[13] IBM Corp. IMS/VS Version 1, General Information Manual, GH20-1260. IBM White Plains, N.Y., Sept. 1980.