Distributed Computing (1987) 2:45–59

DISTRIBUTED COMPUTING

Distributed optimistic concurrency control with reduced rollback

Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta
Department of Computer Science, SUNY at Stony Brook, NY 11794, USA

Divyakant Agrawal *is currently a graduate student in the Department of Computer Science at the State University of New York at Stony Brook. He received his B.E. degree from Birla Institute of Technology and Science, Pilani, India in 1980. He worked with Tata Burroughs Limited, from 1980 to 1982. He completed his M.S. degree in Computer Science from SUNY at Stony Brook in 1984. His research interests include design of algorithms for concurrent systems, optimistic protocols and distributed systems.*

Pankaj Gupta *is currently a graduate student in the Department of Computer Science at the State University of New York at Stony Brook. He received M.S. degree in Electrical Engineering from SUNY at Stony Brook in 1982 and M.S. degree in Computer Science from SUNY at Stony Brook in 1985. His research interests include distributed systems, concurrency control, and databases.*

Arthur Bernstein *is a Professor of Computer Science at the State University of New York at Stony Brook. His research is concerned with the design and verification of algorithms involving asynchronous activity and with languages for expressing such algorithms.*

Soumitra Sengupta *is currently a graduate student in the Department of Computer Science at the State University of New York at Stony Brook. He received his B.E. degree from Birla Institute of Technology and Science, Pilani, India in 1980. He worked with Tata Consultancy Services, from 1980 to 1982. He completed his M.S. degree in Computer Science from SUNY at Stony Brook in 1984. His research interests include distributed algorithms, logic databases and concurrency control.*

Abstract. Concurrency control algorithms have traditionally been based on locking and timestamp ordering mechanisms. Recently, optimistic schemes have been proposed. In this paper a distributed, multi-version, optimistic concurrency control scheme is described which is particularly advantageous in a query-dominant environment. The drawbacks of the original optimistic concurrency control scheme, namely that inconsistent views may be seen by transactions (potentially causing unpredictable behavior) and that read-only transactions must be validated and may be rolled back, have been eliminated in the proposed algorithm. Read-only transactions execute in a completely

Offprint requests to: A.J. Bernstein

This work was supported by the National Science Foundation under grant DCR-8502161 and the Air Force Office of Scientific Research under grant AFOSR 810197

asynchronous fashion and are therefore processed with very little overhead. Furthermore, the probability that read-write transactions are rolled back has been reduced by generalizing the validation algorithm. The effects of global transactions on local transaction processing are minimized. The algorithm is also free from deadlock and cascading rollback problems.

Key words: Database systems – Concurrency – Optimistic protocols – Distributed algorithms – Rollbacks – Transactions

1 Introduction

Concurrency control algorithms [3] have received considerable attention in the literature. Two concurrency control mechanisms – locking [7] and timestamp ordering [16] – can be considered pessimistic in their outlook. To avoid inconsistency, they synchronize at each step of a transaction. Other drawbacks of such algorithms include the deadlock problem in locking and unnecessary rollbacks in timestamp ordering. Recently, optimistic schemes [9] have been proposed for concurrency control. In this approach the optimistic assumption is made that concurrent transactions will rarely conflict and therefore synchronization at each transaction step is wasteful. Instead, transactions are given unrestricted read access to the database during their initial, *read* phase. Transactions write only in local storage during this phase. Transactions then enter a *validation* phase to check for conflicts, and, if successful, read-write transactions enter a *write* phase to incorporate their updates into the database. If validation fails the transaction restarts. This scheme has several drawbacks. Since transactions are not synchronized during their read phase they may see an inconsistent database. Though ultimately aborted, their behavior prior to validation is unpredictable. Secondly, transaction rollback represents wasted work. The frequency with which this may occur in standard optimistic schemes has been criticized [2].

Optimistic concurrency control has been extended for use in a relational database [4]. A proposal has been made to modify the basic optimistic scheme so that read-only transactions do not have to be validated [17], but read-write transactions must still be validated against read-only transactions. Furthermore, the updates of a transaction cannot be installed until there is no conflicting read-only transaction in progress, implying an unbounded delay in the completion of read-write transactions. An attempt to unify optimistic and locking schemes has also been proposed [13]. A distributed optimistic scheme has been proposed by [5] in which dependency graphs are maintained and timeouts are used to avoid deadlocks. Unfortunately, calculation of the dependency graph requires a significant amount of computation.

Multi-version schemes [16, 6] have been proposed to increase concurrency and to reduce transaction rollback by providing transactions with a succession of views of database objects. A multiversion distributed optimistic scheme [10] has been discussed using global timestamps. However, the probability that read-write transactions will be rolled back is the same as in the original optimistic scheme and, in addition, the possibility of cascading rollbacks exists.

This paper proposes a distributed optimistic concurrency control algorithm for use in a multiversion relational database which is based on the original proposal presented in [9]. The proposed scheme is designed to overcome the difficulties of that algorithm. Thus, transactions cannot see inconsistent data and therefore do not behave unpredictably during their read phase. From this it follows that read-only transactions do not need to be validated or rolled back. This makes the algorithm particularly suitable for query dominant systems since the overhead of validation is eliminated for all such transactions and the overall frequency of rollback is greatly reduced. Furthermore, the validation technique has been generalized to reduce the probability of rollback for read-write transactions. The updates of a transaction are stored in intentions lists [12], and are propagated to the database atomically at commit time. The intentions lists serve as the write set of a transaction and thus form an integral part of the concurrency control algorithm as well. Finally, the effects of global transactions on local transaction processing are kept minimal and deadlocks are not possible.

In Section 2 several optimistic algorithms are described which provide concurrency control for the single site case. In Section 3, the algorithms for maintaining multi-version relations are presented. An analysis of rollback is presented in Section 4 and the distributed case is described in Section 5.

2 Single site concurrency scheme

In the standard optimistic concurrency control algorithm, a monotonically increasing *transaction number counter*, (*tnc*), is maintained. When a transaction, T, enters its read phase it takes the value of tnc as its *start number*, *sn(T)*. When it finishes, the transaction counter is incremented and the transaction then takes the value of tnc as its *transaction number*, *tn(T)*. These two numbers are used to delimit the start and finish points of T. Transactions with transaction numbers lying between sn(T) and tn(T) have entered validation while T was executing and may have interfered with T's execution. The test for interference involves checking for non-null intersections between the read and write sets of the transactions. If the simplifying assumption is made that a transaction's write set is contained in its read set, then T conflicts with a prior transaction (with transaction number between sn(T) and tn(T)) if the intersection between the former's read set and the latter's write set is not empty. In this paper we will assume that a transaction's write set is contained in its read set. The algorithms presented can be refined if this is not the case.

In the multi-version scheme presented here, sn(T) is in addition used to select an appropriate version of each relation. The mechanism used to extract the view corresponding to sn(T) is described in Section 3. It is designed so that T does not see the effects of concurrent read-write transactions (whose transaction numbers must be greater than sn(T)). T is guaranteed of seeing a consistent view of the database which is the result of a serial execution of transactions that had already committed when T started (i.e., transactions T′ such that $\mathrm{tn}(T') \leq \mathrm{sn}(T)$). Two important implications follow from this: transactions behave predictably since they operate on consistent views and read-only transactions need not be validated and will not be rolled back. Furthermore, a transaction sees only the results of previously committed transactions and therefore cascading rollbacks cannot occur. A read-write transaction, however, must be validated and may be rolled back since it may have executed concurrently with another read-write transaction whose effect it should have seen.

A set of tuples in a relation can be designated by a predicate which specifies values for some of the attributes. All tuples which agree in these values are in the set. The read set of T, RS(T), consists of the read predicates used during the read phase. The updates of T are not performed directly on the stored relations. Rather, for each relation that is modified an intentions list, or write set WS(T), is maintained which is the log of modifications that T makes to that relation. The modifications consist of: (i) tuples to be inserted in the relation, and (ii) predicates describing tuples to be deleted. An update to a tuple is treated as a delete of the old tuple followed by an insert of a new tuple.

2.1 Serial validation

As in the original optimistic scheme [9], one transaction may be executing its validation or write phases at a time in the multi-version serial validation algorithm (MVSV). The algorithm and a function to check for transaction conflicts are shown in Figs. 1 and 2 respectively. Critical sections are bracketed with "≪" and "≫". There is, in fact, no difference between MVSV and the serial validation of [9], except that read-only transactions are no longer validated. As a result, in the discussion of the algorithms presented here we will be concerned with read-write transactions unless specifically stated otherwise. A read-write transaction T can be successfully validated with respect to a concurrent read-write transaction T′ which has previously committed if the *standard intersection check* is satisfied: $\mathrm{WS}(T') \cap \mathrm{RS}(T) = \phi$. The transaction number is assigned before the write phase so that T's updates can be tagged with tn(T) and thus different versions of a relation can be identified. The transaction counter is incremented after the write phase has completed. The equivalent serial order of transactions is the order of their transaction numbers. Start numbers are assigned the current value of tnc. A transaction, T_1, in its read phase sees the results of all transactions, T_2, such that $\mathrm{sn}(T_1) \geq \mathrm{tn}(T_2)$. Since all such transactions have committed and written to the database, T_1 can be provided with a consistent view.

```
≪
    $tn(T) \longleftarrow tnc+1$;
    IF Validate($T$) THEN
        {$Write(WS(T), tn(T))$;
        $tnc \longleftarrow tnc+1$;}
    ELSE
        {Abort($T$);}
≫
```

Fig. 1. Multi-version serial validation algorithm

```
FUNCTION Validate($T$: TxnDesc): BOOLEAN;
    FOR All $T_i$: $sn(T) < tn(T_i) < tn(T)$ DO
        IF $WS(T_i) \cap RS(T) \neq \phi$ THEN
            {RETURN(FALSE);}
    RETURN(TRUE);
END Validate;
```

Fig. 2. Function validate

2.2 Parallel validation

Serial validation is not attractive since the validation and write phases are done in a single critical section, resulting in a bottleneck. The multi-version parallel validation algorithm (MVPV) allows several transactions to be in their validation or write phases at the same time, and therefore allows greater concurrency.

In the parallel validation algorithm described in [9], the equivalent serial order of transactions is dictated by the order in which they enter the critical section that precedes the validation phase. The transaction numbers, however, are assigned in a different critical section at the end of the write phase. Since transactions may enter the validation phase in a different order than they leave the write phase, the transaction number order does not necessarily correspond to an equivalent serial order. Thus, start numbers cannot be used to obtain consistent views.

To overcome this problem MVPV uses two transaction counters, *ctnc* (commit tnc) and *vtnc* (visible tnc) in order to guarantee that transaction numbers correspond to an equivalent serial order. An *active queue*, AQ, is maintained containing entries for transactions that have entered validation. An entry, $E(T_i)$, of AQ contains, among other things, a transaction identifier field ($E(T_i)$.id), a transaction number field ($E(T_i)$.num) and a type field ($E(T_i)$.type – described below). AQ is ordered on $E(T_i)$.num. MVPV is depicted in Fig. 3.

Vtnc supplies start numbers for transactions. Its value determines the most current, consistent view since it is the largest number satisfying the property that all transactions T_i such that $\text{tn}(T_i) \leq \text{vtnc}$ have committed and finished writing. Ctnc contains the largest transaction number of any transaction that has entered validation. Transaction numbers are assigned the incremented value of ctnc. When a transaction T enters validation it makes a copy of the suffix of AQ starting from the entry $E(T_i)$ with the smallest number field satisfying $E(T_i).\text{num} > \text{sn}(T)$. This determines the transactions against which T validates. An entry for T of type VALIDATING is appended to AQ.

Validation is performed by the function Validate Predecessors, illustrated in Fig. 4. The parameters identify a transaction, T, and an open interval of transaction numbers, [from, to]. The function validates T against all transactions with entries in AQCopy(T) having transaction numbers in the interval. Validation is carried out in *increasing order* of transaction numbers. If all transactions in the given interval can precede T then the function returns null. However, if the function finds a conflict-

```
InitialCS:
≪
    $ctnc \longleftarrow ctnc + 1$;
    $tn(T) \longleftarrow ctnc$;
    Allocate entry $E(T)$;
    $E(T).id \longleftarrow T$;
    $E(T).type \longleftarrow$ VALIDATING;
    $E(T).num \longleftarrow tn(T)$;
    $AQCopy(T) \longleftarrow$ CopyAQ([$sn(T)$, $tn(T)$]);
    InsertAQ($E(T)$, $tn(T)$);
≫
Validation:
    IF (ValidatePredecessors($T$, [$sn(T)$, $tn(T)$]) $\neq NULL$) THEN
        {Abort($T$); EXIT}
WritePhase:
    Write($WS(T)$, $tn(T)$);
≪
    $E(T).type \longleftarrow$ WRITTEN;
    FOR $E(T_i) \in AQ$: $vtnc < E(T_i).num \leq ctnc$ IN INCREASING ORDER DO
        IF $E(T_i).type =$ WRITTEN THEN
            $vtnc \longleftarrow E(T_i).num$;
    ELSE
            EXIT;
≫
```

Fig. 3. Multi-version parallel validation algorithm

```
FUNCTION ValidatePredecessors($T$: TxnDesc; $[from, to]$: TxnNoInterval): TxnNo;
BEGIN
    FOR $E(T_i) \in AQCopy(T)$: $from < E(T_i).num < to$ IN INCREASING ORDER DO
        IF $WS(T_i) \cap RS(T) \neq \phi$ THEN
            {RETURN($E(T_i).num$);}
    RETURN($NULL$);
END ValidatePredecessors;
```

Fig. 4. Function ValidatePredecessors

ing transaction T′ such that $WS(T') \cap RS(T) \neq \phi$, then it returns the value tn(T′), which is the smallest transaction number of any concurrent transaction that cannot precede T in an equivalent serial order. In this case T is aborted. If validation is successful T executes its write phase tagging its updates with tn(T) and then changes the type of its entry in AQ to WRITTEN. The entry $E(T_i)$ in AQ with the largest number satisfying the property that it and all earlier entries are WRITTEN is then located and vtnc is set equal to $E(T_i)$.num. If vtnc cannot be increased then T's updates will not yet be visible to other transactions. Delayed visibility is necessary since otherwise the view of the database determined by the start number of a transaction may not include the updates of all transactions with smaller transaction numbers and hence would be inconsistent. Thus the results of committed transactions are made visible in the order of transaction entry into validation (i.e., the serialization order).

Although the visibility of a transaction may be delayed, the results computed by the transaction may be returned immediately to the user. Because of the delay there is the danger that if T_1 and T_2 are successive transactions in the same process (or in processes that are synchronized so that T_2 must follow T_1), the results produced by T_1 may not be visible to T_2. In order to overcome this, a transaction returns its transaction number to the user on completion, and a user may specify a minimum acceptable start number on transaction initiation. By supplying the transaction number of the preceding transaction as the minimum start number of the succeeding transaction a process can be assured that the latter will see the results of the former. A transaction, thus, may have to be delayed until its requested view is available. Deadlocks are not possible because only transactions in their read phase may wait, and they do not wait for each other. This issue will be discussed again in the section describing the distributed algorithm and a simple implementation of waiting will be presented.

Note that the algorithm would still work correctly if sn(T) were less than the current value of vtnc on entry into the read phase. This would cause T to get an older version of the database, and if T was a read-write transaction, it would make it more vulnerable to rollback (since it will be validated against a larger set of transactions); but it is a degree of freedom which can be exploited when the algorithm is extended to the distributed case. Similarly, ctnc can be incremented by more than one, leaving gaps in the sequence of transaction numbers. This is also useful in the distributed case.

There are two aspects of the algorithm which result in overhead for read-only transactions. The first is that they must initially get a start number. This represents a relatively negligible amount of additional work (particularly since, as noted above, this need not be the most recent value of vtnc). A more important consideration is that all transactions must be sure to access the correct version of each relation, an issue dealt with in the next section. It is important to recognize, however, that in other respects a read-only transaction proceeds at its own pace and does not synchronize with other activities in the database nor perform extra functions related to concurrency control.

If a read-write transaction fails validation, then it must be restarted. The function *Abort* is responsible for discarding the read and write sets of the transaction and deleting its entry from AQ. The transaction restarts itself by obtaining a new start number. For a committed transaction, T, E(T) and WS(T) can be discarded when the probability that there exists a read-write transaction T′ in its read phase such that $sn(T') < tn(T)$ is sufficiently small. Premature disposal will cause unnecessary rollbacks since subsequent transactions will not be able to perform validation. The point at which these data structures are disposed of is an independent policy of the system which will not be discussed in the paper and is not included in the algorithms.

2.3 *Parallel validation, generalized*

The equivalent serial order of transactions determined by MVPV is the order of entry into validation. Since transaction numbers are assigned the incremented value of ctnc at the beginning of the validation phase, the transaction number order corresponds to the serialization order. A validating transaction, T, is aborted if there exists an earlier concurrent transaction whose write set has a non-null intersection with RS(T). It may, however, be possible to commit T in this case using a different transaction number. The reassignment would preserve the correspondence between the equivalent serial order and the transaction number order but the equivalent serial order would no longer correspond to the order of entry into validation. This sub-section presents a generalized parallel validation algorithm (MVGV) which functions in this way. Thus, MVGV is capable of committing read-write transactions that fail validation in MVPV.

Suppose a validating read-write transaction, T, detects a conflict with a concurrent transaction, T′, while performing the standard intersection check: $WS(T') \cap RS(T) \neq \phi$. Then T′ cannot precede T in the equivalent serial order since T should have seen the results produced by T′. In MVPV, T would be aborted. It may be the case, however, that T′ can follow T in an equivalent serial order. This would be true if a *reverse intersection check* is satisfied: $WS(T) \cap RS(T') = \phi$. Thus, the strategy of MVGV is to divide the transactions which are not yet visible but which entered validation before T into predecessors and successors of T. T is assigned a new transaction number which is larger than that of all predecessors and smaller than that of all successors. For all predecessors the standard validation condition must hold, whereas for all successors the reverse validation condition must hold. The justification for this approach rests on the observation that if transactions had entered validation in transaction number order, the reverse intersection checks performed by T would have been done as the standard intersection checks by the successors of T. The only difference in this case is that if the condition is not satisfied then T, not its successor, is rolled back.

If transaction numbers are restricted to be integers it may not be possible to assign a new number to T in the appropriate range which is different from those already assigned to other transactions. This problem can be minimized by allowing transaction numbers to be real numbers. In the following it is assumed that a unique real number can always be assigned. In the rare situation in which this is not possible the transaction can be aborted.

In MVPV all transactions are required to maintain their read and write sets until validation is completed, after which read sets can be discarded but write sets are retained so that subsequent transactions can be validated. In MVGV, RS(T) must be maintained for some time after T has completed validation. However, unlike WS(T), RS(T) can be discarded when $vtnc \geq tn(T)$ since RS(T) is only used by transactions attempting to validate with transaction numbers less than tn(T). But transactions cannot be committed with numbers less than vtnc because their results should already be visible. Thus, this requirement does not impose a significant additional overhead.

The MVGV algorithm is given in Fig. 5. The initial assignment to tn(T) is the incremented value of ctnc at the time of entry into validation. The current values of vtnc and ctnc are copied in the variables lower(T) and upper(T) respectively. T makes a copy of the active queue entries with transaction numbers greater than sn(T) (corresponding to transactions against which it must validate) and then appends an entry for itself.

Validation utilizes two auxiliary functions, ValidatePredecessors and ValidateSuccessors, shown in Figs. 4 and 6 respectively. ValidateSuccessors is an analog of ValidatePredecessors. When given an (open) interval of transaction numbers it checks in *decreasing order* if each of the transactions with entries in AQCopy(T) having transaction numbers in the interval can be successors of T. This is determined by using the reverse intersection check. The function returns null if all transactions in the interval can succeed T. Otherwise, it returns the largest transaction number of the transactions in the interval that cannot succeed T.

Validation is initiated by invoking the function ValidatePredecessors with the interval [sn(T), tn(T)]. The value returned is stored in HighTn(T). If it is null the situation is identical to successful validation in MVPV. Otherwise, the function ValidateSuccessors is invoked with the open interval [$HighTn(T) - \varepsilon$, tn(T)]; ε is the smallest positive real number in the system and is used here so that the transaction whose conflict with T was detected by ValidatePredecessors is included in the open interval passed to ValidateSuccessors. A null in the returned variable LowTn(T) implies that all transactions with entries in AQCopy(T) having transaction numbers greater than or equal to HighTn(T) can be successors of T and a new value, $HighTn(T)^{-}$ [1] is assigned to tn(T). If LowTn(T) is

[1] The notation $HighTn(T)^{-}$ is used to indicate that E(T) is placed immediately before E(T′) in AQ where tn(T′) = HighTn(T). Similarly, the condition immediately after E(T′), will be indicated using +. The exact value of tn(T) is decided upon in the function InsertAQ

```
InitialCS:
  $\ll$
    $ctnc \longleftarrow ctnc + 1$;
    $tn(T) \longleftarrow ctnc$;
    $lower(T) \longleftarrow vtnc$; $upper(T) \longleftarrow ctnc$;
    $AQCopy(T) \longleftarrow$ CopyAQ($[sn(T), tn(T)]$);
    Create entry $E(T)$ for $T$;
    InsertAQ($E(T), tn(T)$);
  $\gg$

Validation:
    $HighTn(T) \longleftarrow$ ValidatePredecessors($T$, $[sn(T), tn(T)]$);
    IF ($HighTn(T) = NULL$) THEN
        GO TO WritePhase;
    ELSE
        {$LowTn(T) \longleftarrow$ ValidateSuccessors($T$, $[HighTn(T) - \varepsilon, tn(T)]$);
        IF $LowTn(T) = NULL$ THEN
            $tn(T) \longleftarrow HighTn(T)^-$;
        ELSE
            {Abort($T$); EXIT}}

ReorderAQ:
  $\ll$
    IF $vtnc \geq tn(T)$ THEN {Abort($T$); EXIT}
    DeleteAQ($E(T)$);
    $E(T).num \longleftarrow tn(T)$;
    $AQCopy(T) \longleftarrow$ CopyAQ($[lower(T), upper(T)]$);
    InsertAQ($E(T), tn(T)$);
  $\gg$

AugmentedValidation:
    IF (ValidatePredecessors($T$, $[lower(T), tn(T)]$) $\neq NULL$) THEN {Abort($T$); EXIT}
    IF (ValidateSuccessors($T$, $[tn(T), upper(T)]$) $\neq NULL$) THEN {Abort($T$); EXIT}

WritePhase:
    – As in Fig. 3 –
```

Fig. 5. Multi-version generalized validation algorithm

```
FUNCTION ValidateSuccessors($T$: TxnDesc; $[to, from]$: TxnNoInterval): TxnNo;
BEGIN
    FOR $E(T_i) \in AQCopy(T)$: $from > E(T_i).num > to$ IN DECREASING ORDER DO
        IF $WS(T) \cap RS(T_i) \neq \phi$ THEN
            {RETURN($E(T_i).num$);}
    RETURN($NULL$);
END ValidateSuccessors;
```

Fig. 6. Function ValidateSuccessors

not null, T is aborted since no satisfactory transaction number for T is possible.

Since AQ is maintained in transaction number order it is necessary to change the position of E(T) in AQ to reflect its new transaction number. Unfortunately, T's position cannot simply be changed without considering the effect of this on other transactions that may also have moved in the interim. This additional computation is performed in the sections labelled ReorderAQ and AugmentedValidation in Fig. 5. An initial check is done to ascertain whether the final value of tn(T) is larger than vtnc. If not, T is aborted since it cannot be inserted in the visible range. Otherwise, T makes a new copy of AQ to be used for the additional validation and reorders its entry in AQ. A simple and straightforward approach which guarantees serializability using the new transaction number is to revalidate T with respect to all transactions in the new copy of AQ, as shown in the Fig. 5. Much of this work has already been performed in the section labelled Validation and so in reality the only additional conditions which must be checked are with respect to transactions that have moved since the initial copy of AQ was made. Such transactions are ones for which conflicts have been detected and hence, if the optimistic assumption is true, only a few additional checks will be required. Since transactions could not have moved into positions before lower(T) (the value of vtnc when T entered validation) this serves as a lower bound on the copy. Similarly, upper(T) (the value of ctnc when T entered validation) serves as an upper bound since a transaction, T′, such that $\mathrm{tn}(T') > \mathrm{upper}(T)$ entered validation after T and either has not moved or, if it has, remains a successor to the initial position of T. It will therefore perform the correct validation checks with respect to T.

It should be noted that read-write transactions which would be committed by MVPV or the parallel validation algorithm in [9] will be committed by MVGV and the same amount of computation will be involved. Such transactions will be committed with their initial assignment of transaction numbers. The effect of MVGV is to reduce rollback. Only transactions that would have been aborted by the former algorithms commit with modified transaction numbers and execute the additional computation contained in the sections of the algorithm labelled ReorderAQ and AugmentedValidation. The same amount of computation is performed in the section labelled Validation whether or not a conflict is detected.

3 Integration and compaction

In this section we propose a technique for storing tuples which allows multiple versions of a relation to be extracted. A two stage process, consisting of *integration* and *compaction*, for merging a write set with a relation is described. The technique has the property that it can be performed concurrently with the execution of other transactions in their read phase without requiring any synchronization.

3.1 Integration

Integration is the process of applying the intentions lists of a transaction to the corresponding relations to get newer versions of the relations without destroying earlier versions. All the versions of a relation are stored in one file. In order to maintain a multi-version relation, create and delete fields are associated with each tuple. If tuple t was created by transaction T then the *create field* of t, $c(t)$, contains tn(T). Similarly, the *delete field*, $d(t)$ of t contains tn(T) if t has been deleted by T, or '∞' if t has not been deleted. Tuple t is visible to transaction T if and only if $c(t) \leq \mathrm{sn}(T) < d(t)$. Each transaction must evaluate this inequality while examining t during its read phase. This constitutes the additional computational overhead of maintaining a multi-version database. Several techniques can be used to minimize the space allocated to each field.[2]

Integration is performed during the write phase of a transaction. Suppose T integrates an intentions list, into relation R. The delete field of each visible tuple in R that unifies with a delete predicate is set to tn(T). These tuples are not actually removed from the file and no change is made to indices used to access R since the tuples may still be in the view of other active transactions. Inserted tuples are appended to the relation with value tn(T) in their create fields and '∞' in the delete fields. In this case the indices are also modified. For example, if the tuples were indexed by a B-tree, then the B-tree is updated to include the key values of the inserted tuples.

Since a goal of the algorithm is to eliminate the need for read-only transactions to synchronize with other transactions, it must be possible to perform integration asynchronously with the execution of transactions in their read phase. But if T_1 is in its write phase at the same time that T_2 is in its read phase then $\mathrm{sn}(T_2) < \mathrm{tn}(T_1)$ and T_2 is therefore reading an earlier version of the database than the one being created by T_1. Thus if t is a tuple being deleted by T_1 and $\mathrm{sn}(T_2) \geq c(t)$, t will be seen by T_2 whether or not $\mathrm{tn}(T_1)$ has been assigned to d(t). Similarly if T_1 adds t then $c(t) > \mathrm{sn}(T_2)$, and therefore t will not be seen by T_2 even if the addition

[2] To avoid storing real numbers a transaction installs its version using the smallest integer greater than or equal to its transaction number and vtnc is assigned the largest integer satisfying the property that all transactions having numbers less than or equal to its value have completed their write phases. Thus, when vtnc is incremented several transactions may become visible simultaneously. Note, however, that if T_2 follows T_1 in the equivalent serial order then the updates of T_2 do not become visible before those of T_1. Furthermore, if 4 bytes are allocated for each transaction number then two 3-byte factor registers can be stored in the header of the relation and only 1 byte is needed for each number in each tuple. Depending upon whether the 1-byte number in the tuple is positive or negative, one of the 3-byte register contents is prepended to construct the entire transaction number

occurs before T_2 reads the relation. Thus integration need not be done in a critical section. Integration consists only of in-place changes to the delete fields of existing tuples and the appending of new tuples. If the underlying system provides an atomic append facility, then multiple integrations can be carried out concurrently.

Indices can also be updated concurrently. An algorithm for updating B-trees [14] exists in which no synchronization is required between a process updating the B-tree via page writes and processes which simply read it. Thus a transaction in its write phase which has appended tuples to a relation can update a B-tree index without synchronizing with transactions in their read phase which are accessing the relation through the B-tree. This technique can be extended to other indexing schemes. Note that there may be tuples belonging to different views of the relation which have the same key value. A simple generalization of B-trees allows the index to have multiple instances of a given key value.

3.2 Compaction

The process of discarding older versions of a relation is called compaction. Tuples that have been deleted by transactions which committed in the more distant past are discarded from the relation during compaction. Since certain earlier views of the relation will no longer be constructible, a relation R must be tagged with a *base transaction number*, *bt(R)*, which corresponds to the earliest version of R that can still be constructed. This must be checked by all transactions when the relation is first opened and represents a small, additional overhead. If T needs access to R and $\mathrm{sn}(T) < \mathrm{bt}(R)$, then T must be restarted with a new (larger) start number. As a result, it is now possible for a read-only transaction to abort, but this effect can be minimized by limiting the frequency with which compaction takes place. Successive aborts of the same transaction, however, are highly unlikely.

Compaction can be done by writing the tuples contained in the versions of a relation to be retained (i.e., tuples t such that d(t) is greater than or equal to the new value of bt(R)) to a new file, creating new indices and switching the file pointer on completion. Thus, it can be performed concurrently with transactions executing their read phase without imposing any synchronization requirements on them. Subsequent readers read the contents of the new file. Transactions reading while the relation is being compacted remain linked to the original file. File space is reclaimed when all readers unlink from that file. Compaction and integration, however, can not be performed concurrently as this may result in lost updates. Thus some synchronization among writers is required.

4 Rollback

The major overhead incurred by optimistic concurrency control algorithms is due to transaction rollback [1, 2]. Since such algorithms detect conflict only after a transaction is run to completion, a significant amount of wasted computation may be incurred. In the multiversion algorithms presented in this paper read-only transactions are never rolled back due to conflicts. Any transaction may be rolled back due to premature compaction, but the probability of this happening can be brought arbitrarily close to zero by increasing the time between successive compactions. The probability of rollback of read-write transactions due to conflicts in MVPV is the same as that in the standard optimistic algorithm. The extent to which MVGV reduces this probability is examined in this section. In addition, a technique for reducing the amount of wasted computation by detecting conflicts during the read phase instead of waiting until validation is discussed. Finally, a technique for preventing successive rollbacks of the same transaction is presented.

4.1 Analysis of rollback in MVPV and MVGV

Let K be the average number of transactions in AQ when T enters validation and let T_i be the i^{th} such transaction. Transactions are assumed to be independent of one another. A conflict occurs between T and T_i if either $\mathrm{WS}(T_i) \cap \mathrm{RS}(T) \neq \phi$, which shall be referred as a forward conflict, or $\mathrm{WS}(T) \cap \mathrm{RS}(T_i) \neq \phi$. The probability of either of these events is p and they are assumed to occur independently. T can be successfully committed as the successor of the K transactions in AQ provided it does not have a forward conflict with any of them. The probability that this occurs, $P_{after(K)}(K)$, is:

$$P_{after(K)}(K) = \prod_{i=1}^{K} \mathrm{Probability}(WS(T_i) \cap RS(T) = \phi)$$
$$= (1-p)^K.$$

This is the probability of successful validation in MVPV given that there are K transactions in the queue. Therefore

$$P^{MVPV}(K) = (1-p)^K.$$

If T has a forward conflict with a transaction T_{l+1}, then it cannot commit after T_{l+1} or subsequent transactions. The probability that T can commit as the successor of the first l transactions but not as the successor of the first $l+1$ transactions, $P_{after(l) \wedge \neg after(l+1)}$, is given by

$$P_{after(l) \wedge \neg after(l+1)}$$
$$= \left[\prod_{i=1}^{l} \text{Probability}(WS(T_i) \cap RS(T) = \phi)\right] *$$
$$[\text{Probability}(WS(T_{l+1}) \cap RS(T) \neq \phi)] *$$
$$\left[\prod_{i=l+1}^{K} \text{Probability}(WS(T) \cap RS(T_i) = \phi)\right]$$
$$= p(1-p)^K.$$

$P_{after(i) \wedge \neg after(i+1)}$ and $P_{after(j) \wedge \neg after(j+1)}$ are the probabilities of disjoint events for $i \neq j$. In MVGV, T cannot be placed before any transaction that has been made visible during its read phase. If k of the K transactions are visible then the probability that T will successfully validate in MVGV is given by

$$p^{MVGV}(K)$$
$$= \left[\sum_{i=k}^{K-1} P_{after(i) \wedge \neg after(i+1)}(T)\right] + P_{after(K)}(K)$$
$$= \left[\sum_{i=k}^{K-1} p(1-p)^K\right] + (1-p)^K$$
$$= (1-p)^K(1+(K-k) \cdot p).$$

Thus, the ratio of the probability of success in MVGV to the probability of success in MVPV is $(1+(K-k) \cdot p)$. The probability of rollback as a function of K is plotted for two different values of p in Figs. 7 and 8. The MVGV curve assumes k = 0 and represents the best improvement possible. When k = K, the probability of rollback in MVGV is same as in MVPV. In general, the operating range of MVGV will be between these two extremes. The figures indicate that this range is singificant. Thus, transaction rollbacks in MVGV will almost always be less than in MVPV.

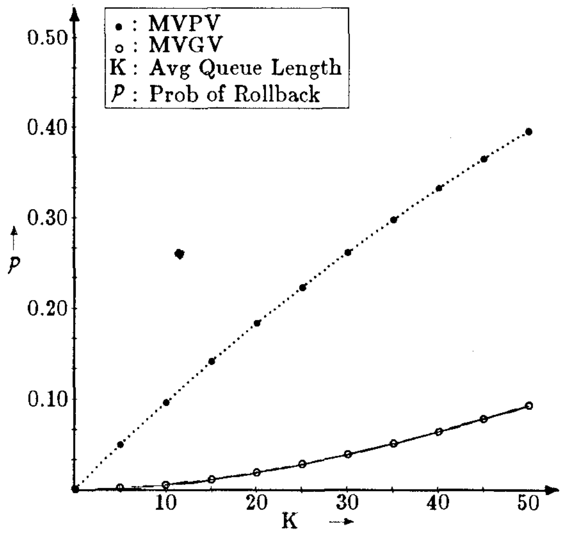

Fig. 7. Probability of rollback versus average queue length for $p = 0.01$

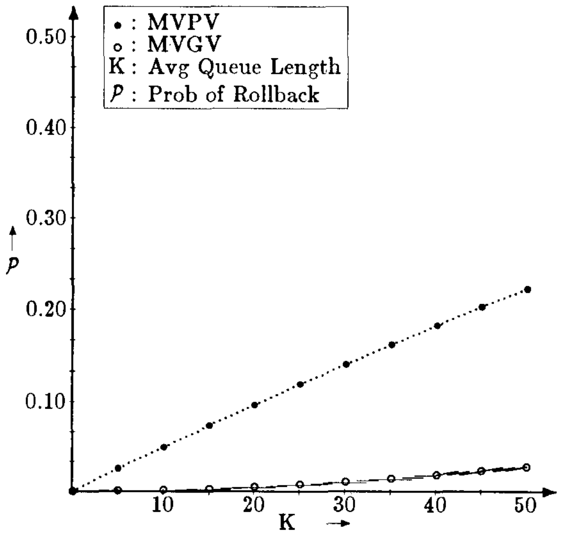

Fig. 8. Probability of rollback versus average queue length for $p = 0.005$

4.2 Immediate rollback

The create and delete fields of a tuple can be used by a read-write transaction to detect a potential conflict while it is still in its read phase. It can then elect to abort itself immediately. Thus, if a read-write transaction T encounters a tuple t satisfying its read predicate and c(t) > sn(T), then t was appended by a prior, but concurrent, transaction, T′, against which T will ultimately validate. Since t is an element of WS(T′), it follows that $WS(T') \cap RS(T) \neq \phi$. As a result, T cannot be a successor of T′ and validation reduces to checking if T can precede T′. Thus the probability of successful validation is significantly reduced and immediate rollback may be deemed appropriate. A similar condition exists with respect to d(t).

4.3 *Successive rollback*

If an aborted transaction, T, is simply restarted as a new transaction then it may be aborted again and, although the probability is quite small, this pattern may repeat an arbitrary number of times. A simple technique for avoiding this exists if T's read and write sets do not change when it is rerun. The essential idea is to force a situation in which the second time T is executed the set of transactions against which it must validate is empty, thus ensuring successful validation.

For example, suppose T is aborted while executing the section labelled Validation in the MVGV algorithm shown in Fig. 5. On its second iteration tn(T) is left unchanged and sn(T) is set equal to $\mathrm{tn}(T)-\varepsilon$. This guarantees that when T enters validation for the second time no transaction will be found with transaction number in the interval [sn(T), tn(T)] and validation will be successful. The scheme can be implemented by simply leaving T's entry in AQ when validation fails, thus using the old transaction number again. Since RS(T) and WS(T) are known at this time, transactions which subsequently enter their validation phase can validate against T despite the fact that T may not have completed its (second) read phase. Thus, validation of other transactions is not effected. Since vtnc may be less than sn(T) when T is to be restarted, it may be necessary to delay the restart. A mechanism for doing this is already an integral part of the distributed version of MVGV (described in the next section) and presents no problem. The drawback of this technique is that while T is being rerun new views cannot be made visible. T is restarted when the previous transaction in AQ completes its write phase and vtnc cannot be advanced until T completes its write phase.

5 Distributed optimistic concurrency control

The single site algorithm extends naturally to a distributed environment. An agent process called the *transaction server* executes at each site and is responsible for handling the validation of local transactions as well as the sub-transactions of global transactions at that site. All servers maintain their own local counters. Distributed concurrency control algorithms generally guarantee the atomicity of global transactions by employing a commit protocol [8, 15]. The server at the coordinator site of a global transaction initiates the commit protocol at the end of the read phase, and the servers at the cohort sites validate the sub-transactions. If the initial phase of the commit protocol is successful, (i.e., all cohorts reply in the affirmative to the validation request), the coordinator starts the final phase of the protocol during which the cohorts integrate the respective intentions list at each site. Each cohort is concerned only with the local read and write sets of the global transaction at that cohort site.

The extension of the single site parallel validation algorithm to a distributed environment should not only guarantee the serializability of local and global transactions but, to eliminate the need to validate global read-only transactions, should guarantee globally consistent views to all transactions. Unfortunately, a naive extension of the algorithm in which sub-transactions of a global transaction are independently validated without coordination among the cohort sites leads to non-serializable execution schedules as well as to inconsistent views. The following examples illustrate these problems.

Suppose two global transactions, T_1 and T_2, access relations at sites S_1 and S_2. Let relation R_1 be on S_1 and relation R_2 be on S_2. The read sets of T_1 and T_2 are both $\{R_1, R_2\}$, the write set of T_1 is $\{R_2\}$ and the write set of T_2 is $\{R_1\}$. If validation of T_1 and T_2 at sites S_1 and S_2 are such that $\mathrm{tn}(T_1)<\mathrm{tn}(T_2)$ at S_1 and $\mathrm{tn}(T_2)<\mathrm{tn}(T_1)$ at S_2 then both transactions will commit despite the fact that if both relations had been on the same site one of the transactions would have been aborted.

Similarly, suppose a global transaction, T_1, commits at sites S_1 and S_2, and the write phase of T_1 at S_1 finishes before the write phase at S_2. If the vtnc at S_1 is advanced and at this instant another global transaction, T_2, obtains start numbers from S_1 and S_2, the relations read by T_2 from S_1 will reflect the completion of T_1 whereas the relations read from S_2 will not. Thus, T_2 will have an inconsistent view of the database. Note that these problems are limited to global transactions.

The distributed algorithm described here guarantees the serializability of a global transaction by requiring that its sub-transactions commit with the same transaction number at all cohorts. This ensures that if there are two concurrent global transactions involving common cohorts, then at each such site the serial order of the two transactions will be same. The transaction number assigned to a global transaction is called a *global transaction number* (*gtn*).

Similarly, the problem of inconsistent views can be eliminated by requiring that a global transaction uses the same start number at all cohorts. A global transaction gets a start number from the coordinator and uses it to extract a view from all other sites. This requirement, in conjunction with the property that a global transaction commits with the same transaction number at all sites, ensures a consistent view to a global transaction. The start number assigned to a global transaction is referred to as its *global start number* (*gsn*).

```
≪
    IF ($gsn(T) > vtnc$) THEN
        {$ctnc \longleftarrow$ MAX($gsn(T)$, $ctnc$);
        Allocate entry $E(T)$;
        $E(T).id \longleftarrow T$;
        $E(T).type \longleftarrow$ WAITING;
        $E(T).num \longleftarrow gsn(T)$;
        InsertAQ($E(T)$, $gsn(T)$);
        Exit Critical Section and Wait}
≫
```

Fig. 9. Initiating a sub-transaction at a cohort site

5.1 *Algorithm*

The details of a distributed algorithm based on MVGV are presented here. Since the treatment of local transactions is unchanged, only considerations pertaining to global transactions will be discussed and dealt with in the figures. Note, however, that ctnc, vtnc, and AQ are used for both the validation of local transactions as well as sub-transactions of global transactions. Validation of a sub-transaction at a cohort site involves only the local read and write sets at that site. The distributed

```
receive($coordinator$, $request$, $T$, $gtn(T)$);
InitialCS:
≪
    $ctnc \longleftarrow$ MAX($gtn(T)$, $ctnc$);
    $tn(T) \longleftarrow$ MAX($gtn(T)$, $vtnc^+$);
    $lower(T) \longleftarrow vtnc$; $upper(T) \longleftarrow ctnc + \varepsilon$;
    $AQCopy(T) \longleftarrow$ CopyAQ([$sn(T)$, $upper(T)$]);
    Create entry $E(T)$ for $T$;
    InsertAQ($E(T)$, $tn(T)$);
≫
Validation:
    $HighTn(T) \longleftarrow$ ValidatePredecessors($T$, [$sn(T)$, $tn(T)$]);
    IF ($HighTn(T) = NULL$) THEN
        {$LowTn(T) \longleftarrow$ ValidateSuccessors($T$, [$tn(T)$, $upper(T)$]);
        IF $LowTn(T) = NULL$ THEN
            GO TO SendVote;
        ELSE
            {$HighTn(T) \longleftarrow$ ValidatePredecessors($T$, [$tn(T)$, $LowTn(T) + \varepsilon$]);
            IF ($HighTn(T) = NULL$) THEN
                $tn(T) \longleftarrow LowTn(T)^+$;
            ELSE
                {Abort($T$); EXIT}}}
    ELSE
        {$LowTn(T) \longleftarrow$ ValidateSuccessors($T$, [$HighTn(T) - \varepsilon$, $upper(T)$]);
        IF $LowTn(T) = NULL$ THEN
            $tn(T) \longleftarrow HighTn(T)^-$;
        ELSE
            {Abort($T$); EXIT}}
ReorderAQ:
    – As in Fig. 5 –
AugmentedValidation:
    – As in Fig. 5 –
SendVote:
    send($coordinator$, VALID, $T$, $tn(T)$);
```

Fig. 10. Distributed multi-version generalized validation algorithm

Coordinator | Cohorts

Phase I

Coordinator:

```
$gtn(T) \longleftarrow$ MAX(all ctnc) $+ \Delta$;
send(cohorts, VALIDATE, T, gtn(T));
```

Cohorts:

```
– As in Fig. 10 –
```

Phase II

Coordinator:

```
EvaluateVote:
receive(cohorts, replies, T, tn);
CASE replies OF
  Any cohort replies INVALID:
    send(cohorts, ABORT, T, NULL); EXIT;
  All cohorts reply VALID:
    $gtn_{min}(T) \longleftarrow$ MIN(all tn);
    $gtn_{max}(T) \longleftarrow$ MAX(all tn);
    CASE
      $gtn_{min}(T) = gtn_{max}(T)$:
        send(cohorts, COMMIT, T, NULL); EXIT;
      $gtn_{min}(T) < gtn(T) < gtn_{max}(T)$:
        send(cohorts, ABORT, T, NULL); EXIT;
      $gtn_{min}(T) < gtn(T) \wedge gtn_{max}(T) \leq gtn(T)$:
        send(cohorts, REVALIDATE, T, $gtn_{min}(T)$);
      $gtn_{max}(T) > gtn(T) \wedge gtn_{min}(T) \geq gtn(T)$:
        send(cohorts, REVALIDATE, T, $gtn_{max}(T)$);
    END; (* CASE *)
    GO TO EvaluateVote;
END; (* CASE *)
```

Cohorts:

```
receive(coordinator, request, T, newgtn(T));
CASE request OF
  ABORT: Abort(T); EXIT;
  COMMIT: GO TO WritePhase;
  REVALIDATE:
    IF $newgtn(T) = tn(T)$ THEN
      GO TO SendVote;
    ELSE
      {$tn(T) \longleftarrow newgtn(T)$;
      GO TO ReorderAQ; }
END; (* CASE *)

WritePhase:
          – As in Fig. 3 –
```

Fig. 11. Validation and commit phase of the distributed algorithm

algorithm is shown in Figs. 9–11. The subscripts of ctnc, vtnc and AQ that identify variables at a specific cohort site have been omitted in the figures for a clearer presentation.

If the value of $vtnc_i$ at each cohort, S_i, of a global transaction, T, were known by the coordinator before it began its read phase, then it would be reasonable to assign the minimum of these values to gsn(T) and thus be assured that the view required by each sub-transaction would be available at all cohort sites. In the absence of such information, gsn(T) is assigned the current value of vtnc at the coordinator site. (An assignment of a smaller value is also possible.) The initiation of a subtransaction is shown in Figure 9. If $gsn(T) \leq vtnc_i$ the sub-transaction can proceed immediately since the desired view is available. If $gsn(T) > vtnc_i$ execution at S_i must be delayed until all transactions in their validation or write phases at S_i having transaction numbers less than or equal to gsn(T) have terminated. For this purpose an AQ entry is created for T with E(T).type set to WAITING and E(T).num set to gsn(T). Since AQ is ordered on the number field, when $vtnc_i$ reaches E(T).num the sub-transaction can be started and the entry deleted. A minor modification of the WritePhase section of Fig. 3 is required for this purpose. If $ctnc_i < gsn(T)$ then $ctnc_i$ is set equal to gsn(T). This reduces the probability that subsequent transactions will enter validation with transaction numbers less than gsn(T), forcing T to wait for their completion.

In order to facilitate agreement among cohorts on a common transaction number the coordinator proposes a value for gtn(T) at the end of the read phase and transmits it to all cohorts. Each cohort attempts to validate its sub-transaction using a modified form of MVGV in which the initial assignment to tn(T) is gtn(T) instead of the incremented value of ctnc at that site. Since the attempt may fail, one or more sites may find it necessary to choose a different transaction number. Thus, even if all sub-transactions are successfully validated, additional coordination may be required to generate a common transaction number at all sites. If all cohorts successfully validate using the initial value of gtn(T) then this additional coordination can be avoided. Thus, care must be taken in choosing the initial transaction number. Local transactions do not enter the algorithm with a preassigned transaction number. The incremented value of ctnc is used in that case.

If the coordinator selects a value for gtn(T) which is smaller than vtnc_i at some cohort site then it will not be possible to commit T at that site using the initial assignment, since T cannot be committed with a number which is already in the visible range. Thus, to reduce the probability of reassignment, the initial value of gtn(T) must be larger than vtnc_i at all cohorts. On the other hand, the equivalent serial order should approximately correspond to the order in which transactions are processed in real time. Specifying a value of gtn(T) which is very large not only uses up the number space from which transaction numbers are drawn, but increases the probability that the equivalent serial order of global transactions will differ greatly from the order in which they were initiated in real time. As a result, the coordinator bases its choice of gtn(T) on the values of ctnc_i which it obtains from the cohorts. The initial value of gtn(T) is selected by adding a *safety factor*, Δ, to the maximum of these values. The safety factor is heuristically chosen to reflect the number of transactions that may have entered validation at S_i since ctnc_i was sampled. The goal of the heuristic is to make gtn(T) close to ctnc_i at each cohort. In this way the transaction number assigned to the sub-transaction approximates the number that would have been assigned by the cohort had the sub-transaction been local. It is useful in this regard to keep the commit transaction counters at all sites in rough synchronization. Since these counters can always be incremented, a mechanism similar to that proposed in [11] can be used to exchange and update values. A side effect of this is that the view transaction counters will be kept roughly synchronized as well, which is desirable since it reduces waiting and ensures current views to sub-transactions. Note that an optimal algorithm does not eliminate the possibility of reassignment, which may still be necessary due to conflicts detected at cohort sites.

The distributed MVGV algorithm used to validate a sub-transaction at a cohort site is shown in Fig. 10. The initial critical section first ensures that ctnc_i refers to the largest entry in AQ. If the strategy used by the coordinator in choosing gtn(T) is successful it will be larger than vtnc_i and can serve as the initial assignment to tn(T). If not, another value must be chosen and additional coordination will be required.

The Validation section of the algorithm determines whether T can be validated with the initial assignment and, failing that, whether a different number can be found which allows successful validation. If neither is possible, T must be aborted. In order to be successful with the initial assignment all transactions with entries in AQCopy(T) having transaction numbers smaller than tn(T) must precede T in the equivalent serial order and all transactions with larger numbers must follow T. Validation proceeds accordingly by invoking the function ValidatePredecessors with the open interval [sn(T), tn(T)] and, if a null value is returned, invoking ValidateSuccessors with the open interval [tn(T), upper(T)]. If the returned value is also null then validation of T succeeds with transaction number tn(T). If a non-null value is returned by both functions then conflicts are indicated which require the rollback of T. An INVALID indicator is sent to the coordinator as part of Abort(T). If a single non-null value is returned then a new transaction number is determined with which T can be validated. The number will be smaller than tn(T) if the collision was detected by the original call to ValidatePredecessors and will be larger if the collision was detected by the call to ValidateSuccessors. In the former case the new number is the maximum that can be used for validating the sub-transaction at that cohort; in the latter case it is the minimum. If validation is successful, the transaction number is returned to the coordinator. This constitutes the first phase of the two-phase commit protocol. The cohort then awaits a response from the coordinator before entering its write phase. Note that only global transactions encounter such a wait and that such transactions do not delay each other or local transactions, since a parallel validation algorithm is used.

The coordinator enters the second phase of the commit protocol, shown in Figure 11 after receiving replies from all cohorts. It sends an abort message if any one of the cohorts replies INVALID. If all cohorts reply VALID then the coordinator computes the minimum and maximum, $gtn_{min}(T)$ and $gtn_{max}(T)$, of the transaction numbers returned. If the two are identical, indicating that all sub-transactions have been validated with the same transaction number, the coordinator commits the transaction with that number and sends a commit message to the cohorts. On the other hand, if $gtn_{min}(T)$ is smaller, and $gtn_{max}(T)$ is greater, than $gtn(T)$ then the coordinator sends an abort message to the cohorts. This is necessary because in this case one cohort cannot commit its sub-transaction with a transaction number greater than $gtn_{min}(T)$ and another cannot commit with a number less than $gtn_{max}(T)$. The other two cases arise when $gtn_{min}(T)$ and $gtn_{max}(T)$ are not identical and either both are less than or equal or both are greater than or equal to $gtn(T)$. In the former case all cohorts have either successfully validated with the initial assignment or have determined upper bounds on possible transaction numbers. The latter case is similar except the bounds are lower bounds. In both cases a common transaction number is possible. The coordinator requests the cohorts to revalidate T with $gtn_{min}(T)(gtn_{max}(T))$ as the final global transaction number.

Upon receiving a reply from the coordinator the cohort either aborts, commits or continues validating the sub-transaction with the transaction number returned. In the latter case the cohort reenters validation in such a way that only the new transaction number is acceptable. Hence, one extra phase is required to determine if the transaction can be committed.

6 Conclusions

In this paper a distributed, multi-version optimistic concurrency control scheme has been proposed which is particularly advantageous in query-dominant systems. The algorithm attacks several deficiencies of the original optimistic concurrency control scheme. Since all transactions are guaranteed of seeing a consistent view of the database, unpredictable behavior during the read phase is eliminated. The amount of wasted computation is reduced by (essentially) eliminating rollback for read-only transactions, eliminating the need to validate read-only transactions and generalizing the validation conditions so as to reduce the probability of rollback for read-write transactions. Unlike multi-version timestamp ordering algorithms, read-only transactions cannot cause read-write transactions to abort. The algorithm does not suffer from the deadlock and cascading rollback problems of other distributed optimistic concurrency control schemes.

Acknowledgement. The authors wish to thank Josyula Rao for his helpful comments and criticism.

References

1. Agrawal R, Carey MJ, Livny M (1985) Models for studying concurrency control performance: Alternatives and implications. Proc ACM-SIGMOD, Texas (May 1985) pp 108–121
2. Agrawal R, Dewitt DJ (1985) Integrated concurrency control and recovery mechanisms: Design and performance evaluation. ACM Trans Database Syst 4:529–564
3. Bernstein PA, Goodman N (1981) Concurrency control in distributed database systems. Comput Surv 2:185–221
4. Bragger RP, Reimer M (1983) Predicative Scheduling: Integration of Locking and Optimistic Methods. Tech Rep Eidgenössische Technische Hochschule (ETH), Zürich (July 1983)
5. Ceri S, Owicki S (1982) On the use of optimistic methods for concurrency control in distributed databases. Proc 6th Berkeley Workshop in distributed data management and computer networks (Feb 1982) pp 117–129
6. Chan A, Fox S, Lin WK, Nori A, Ries DR (1982) The implementation of an integrated concurrency control and recovery scheme. Proc ACM-SIGMOD, Florida (July 1982) pp 184–191
7. Eswaran KP, Gray JN, Lorie RA, Traiger IL (1976) The notion of consistency and predicate locks in database system. Commun ACM 11:624–633
8. Gray JN (1978) Notes on data base operating systems. In: Bayer R, Graham RM, Seegmuller G (eds) Operating systems. An advanced course. Lect Notes Comput Sci, vol 60. Springer, Berlin Heidelberg New York pp 393–481
9. Kung HT, Robinson JT (1981) On optimistic methods for concurrency. ACM Trans Database Syst 2:213–226
10. Lai MY, Wilkinson WK (1984) Distributed transaction management in JASMIN. Proc 10th Int Conf on very large data bases, Singapore (August 1984) pp 466–470
11. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 7:558–565
12. Lampson BW, Sturgis HE (1979) Crash recovery in a distributed data storage system. Computer Science Laboratory, Xerox Palo Alto Research Center
13. Lausen G (1982) Concurrency control in database systems: A step towards the integration of optimistic methods and locking. Proc ACM Annual Conf, Dallas, TX (Oct 1982) pp 64–68
14. Lehman PL, Yao SB (1981) Efficient locking for concurrent operations on B-trees. ACM Trans Database Syst 4:650–670
15. Mohan C, Lindsay B (1985) Efficient commit protocols for the tree of processes model of distributed transactions. ACM Operating Syst Review 2:40–52
16. Reed DP (1978) Naming and Synchronization in a Decentralized Computer System. Tech Rep MIT/LCS/TR-205, MIT (September 1978)
17. Schlageter G (1981) Optimistic Methods for Concurrency Control in Distributed Database Systems. Proc 7th Int Conf on very large data bases, Cannes (September 1981) pp 125–130