

ON THE CORRECTNESS OF A LOCAL STORAGE SUBSYSTEM
(Extended Abstract)

Marco A. Casanova*, Arnaldo V. Moura** and Luiz Tucheran***

*Brasilia Scientific Center

**Software Technology Center

***Latin American Systems Research Institute

IBM Brazil

ABSTRACT

The design of a provably correct local storage subsystem that integrates local reliability and concurrency control is analysed. The description of the reliability and concurrency control strategies, as well as the correctness proof, is divided into successive layers of abstraction. The reliability correctness criterion adopted requires that the last certified version and all saved versions of each page always be safely kept by the subsystem. The concurrency correctness criterion is an adaptation of multi-version serializability that reflects reliability requirements.

1. INTRODUCTION

This paper analyses the design of a provably correct local storage subsystem of a database system that integrates reliability and concurrency control. The major contributions of the paper lie in the compartmentalization of the problem and in the way verification techniques are combined to obtain a correctness proof.

The subsystem implements the usual local reliability control functions, such as local roll-back of uncommitted transactions, recovery of the local database from primary failures, and checkpoint of a transaction, which have been studied in the literature [BGH,Gr,GMBL,Hal,Ha2,HR,Ko,Li,RH,Ve]. In the full paper, the creation of a dynamic fuzzy dump of the local database is also discussed. But it also offers enough functions to allow the implementation of the strict two-phase commit protocol. Presently, it does not address global reliability problems, such as replica control [BG2,CFLNR,GSDFR,TGGL] and reintegration of a down site into the DDBMS when it recovers [ABG,HS,PPRS].

To analyse the correctness of the subsystem,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

standard multi-version concurrency control theory [BG1], as well as techniques from concurrent program verification (specially invariance methods [Ke,La,Ow]), were used. Although not adopted at the present stage of development, multilevel atomicity arguments seem to be useful for the kind of analysis proposed [BBGLS,Be,Ly].

But the major obstacle in the development of a correct storage subsystem lied in the interplay between reliability control, concurrency control and efficient access to secondary storage (including buffer management). The complexity of the correctness proof directly reflects these problems, to the point of losing credibility and raising methodological concerns about its validity. To cope with this serious question, we organized the description and analysis of the subsystem into layers of abstraction.

The first layer gives a very high-level description of the reliability and concurrency control strategies adopted. The introduction of this level has several advantages. First, it permitted to precisely state what the reliability and concurrency control criteria are, and to clearly indicate how reliability and concurrency control interact, without getting tangled with implementation details. Second, it lead to a proof that the strategies behave correctly if the computation is stopped at any point, on the assumption that all data structures are non-volatile and that very high-level assignments are atomic actions. That is, it separated the problem of deciding in which order the data structures must be updated (such as recording that a transaction changed state before or after updating the set of versions kept) from the problem of main memory volatility. Finally, the description at this level was detailed enough to avoid vaguenesses about the strategies adopted.

The description of the storage subsystem at this stage uses multiple versions both for reliability and, in part, for concurrency control. It can be viewed as an abstraction of many such strategies (see, e.g., [BHR,BG1,PK,SR]) and, in particular, of that described in [CFLNR]. Other integrated solutions to the reliability and concurrency control problems have been studied, for example, in [Be,Re].

The second level of description refines the first one as far as bringing up reliability issues caused by primary failures (loss of main memory) and by

©1985 ACM 0-89791-153-9/85/003/0123 \$00.75

I/O errors. Primary failures are modelled by classifying data structures as volatile or stable, according to whether they survive primary failures or not, and I/O errors are captured by altering the semantics of assignments to stable data structures. Buffer pool management also becomes visible to the point of explicating how logical writes work. However, refinements that have to do with efficient management of versions, and not with reliability or concurrency control, were avoided. Thus, a third level of refinement (which is not covered in this extended abstract) is still necessary to produce a complete design of the storage subsystem.

Many refinements of the first-level description are possible. The one outlined in this extended abstract was motivated by an analysis of Lorie's shadow page algorithm [Lo], which proved not to be appropriate in the presence of concurrent transactions, as pointed out in [GMBL], since it saves and restores data on a per file basis, whereas this should be done on a per transaction basis. In fact, this problem forced the use of shadow pages and a classical log strategy in the recovery manager of System R [GMBL]. The implementation outlined combines timestamped versions and a log strategy differently. Essentially, the log is used to safely record the current state of each transaction (ACTIVE, COMMITTED, etc.) and, for each version v , the transaction that created v and the disk sector where v resides (note that the value of a version is not kept in the log, just information on where the version resides, which results in a very compact log).

The paper is organized as follows. Section 2 describes the relationship between the storage subsystem and the rest of the DDBMS that is assumed throughout the paper. Section 3 gives a high-level description of the subsystem. Section 4 addresses the correctness problem at the first level of abstraction. Section 5 discusses implementation issues and further correctness issues. Finally, Section 6 contains conclusions.

2. ENVIRONMENT OF THE STORAGE SUBSYSTEM

In this section we describe both the environment and a model of the input of the storage subsystem.

When a transaction is submitted to the system, it passes queries and updates over the global schema, as well as other commands, to the Transaction Manager (TM). The TM first transforms each query or update over the global schema into a parallel program P consisting of data transfer operations, and queries and updates over the various local schemas. The TM also controls the execution of P and, in particular, guarantees that either the transaction commits at all sites that it visited or at none of them (we assume that the two-phase commit protocol is used for this purpose). The local Data Manager (DM) transforms local queries and updates into operations of the Storage Subsystem (SS). The local DM processes requests on behalf of several transactions at the same time, passing operations from different transactions to the local SS.

The local SS translates each storage operation into a sequence of low level operations over the internal data structures of the database. The communication

between the local DM and the local SS is done by sending the following messages:

NAME	MESSAGE	INFORMAL DESCRIPTION
Begin	$B(T,t)$	signals the beginning of transaction T and assigns t as the begin timestamp of T
Read	$R(j,T,k)$	k^{th} read of a version of page j for T
Write	$W(j,T,k)$	k^{th} write of T that creates a new version of page j , or updates the version of j already created by a previous write of T
Save	$S(T)$	saves the updates of T
Commit	$C(T,t')$	commits the updates of T and assigns t' as the commit timestamp of T
Abort	$A(T)$	aborts T , rolling back T 's updates
Restore	RE	restores the database after a primary failure

Note: t and t' are non-negative integers.

At the level of the SS, the behavior of a transaction is modelled by a transaction schedule, which is a sequence of operations that starts with $B(T,t)$, continues with reads and writes, which may be executed concurrently (if not related by an ordering) or sequentially (if they are related by an ordering) and either terminates abnormally with $A(T)$, or correctly with $S(T)$ followed by $C(T,t')$.

More precisely, let $\tau = \{T_1, \dots, T_n\}$ be a transaction set and $\pi = \{1, \dots, M\}$ be a page set. We define a transaction schedule for a transaction T in τ and page set π as a pair $L = (M, <)$ such that:

- $M = \{B(T,t)\} \cup R \cup W \cup E$ where
 - $R \subseteq \{R(j,T,k) / j \in [1,M], k > 0\}$
 - $W \subseteq \{W(j,T,k) / j \in [1,M], k > 0\}$
 - E is one of the following five sets: \emptyset , $\{S(T)\}$, $\{S(T), C(T,t')\}$, $\{A(T)\}$, or $\{S(T), A(T)\}$
- if $B(T,t)$ and $C(T,t')$ are in M then $t < t'$
- $< \subseteq M \times M$, the message order of L , is a partial order over M such that:
 - $B(T,t)$ is the minimum of M w.r.t. $<$
 - if $A(T) \in M$ then $A(T)$ is the maximum of M w.r.t. $<$
 - if $S(T) \in M$ then $S(T)$ is the maximum of $M - \{C(T,t'), A(T)\}$ w.r.t. $<$
 - if $C(T,t') \in M$ then $C(T,t')$ is the maximum of M w.r.t. $<$

If M contains either $C(T,t')$ or $A(T)$ we say that L is complete, otherwise we say that L is partial.

The input to the storage subsystem is modelled by a system schedule, which is a set S of transaction schedules (for transaction set τ and page set π) such that S is a set of transaction schedules of distinct transactions in τ . This restriction implicitly says that a transaction in τ cannot "be executed twice", and is just a notational convenience.

nience.

Finally, for concurrency control reasons, transactions are classified as queries and updates. A query is a transaction that only reads from the database, that is, it issues no write messages. For simplicity, we assume that queries continue to issue save and commit messages, although this is really not necessary. An update transaction reads and writes from the database. We assume that the type of a transaction is known when the transaction begins.

3. HIGH-LEVEL DESCRIPTION OF THE ALGORITHM

We present in this section a high-level description of the operations of the storage subsystem. We first informally discuss how messages are input to the storage subsystem, then we introduce the (abstract) data structures used by the operations and, finally, we describe the operations themselves.

The upper level routines of the database system send a stream of messages to the storage subsystem that is modelled by a system schedule S , as described in Section 2. For each transaction log $L=(M, <)$ in S , the storage subsystem selects messages in M for processing according to the ordering dictated by the relation $<$. That is, if $0 < 0'$ then 0 must complete processing before $0'$ is selected for processing. However, the processing of a message may be blocked for concurrency control reasons and, symmetrically, the processing of a message may unblock one or more messages, that start to compete for processing together with the new incoming messages. These aspects of the storage subsystem will not be discussed further in this extended abstract.

The high-level specification of the storage subsystem operations is based on certain abstract objects, classified as proof variables, used for specification and correctness purposes only, and as program variables, which are directly manipulated by the operations:

PROOF VAR.	INTERPRETATION
τ	set of pairs $T=(I, Y)$, where I is the transaction id and Y is the transaction type, which is either <u>Update</u> or <u>Query</u>
π	an integer interval $[1, M]$ representing the set of pages in question
ρ	set of messages processed thus far, excluding those that were blocked. A write message $W(j, T, k)$ is represented simply as $W(j, T)$. Likewise, a read message $R(j, T, k)$ is represented as $R(j, T)$.
ω	a function giving the clock value when a save message began processing

Note: if $T=(I, Y) \in \tau$, we use $T.id$ and $T.type$ to denote I and Y , respectively.

PROGRAM VAR.	INTERPRETATION
CLOCK	a local clock taking values from the set N of naturals
STATE	a function from τ into the set $\{IDLE, ACTIVE, COMMITTING, ABORTED, COMMITTED\}$
VERSION_POOL	a set of pairs $(j, T) \in \pi \times \tau$ representing page versions kept by the system. The value of a version plays no role here and is, therefore, ignored
BTIME	a partial function from τ into N such that $BTIME(T)=t$ iff $B(T, t) \in \rho$ (i.e., $B(T, t)$ was already processed)
CTIME	a partial function from τ into N such that $CTIME(T)=t$ iff $C(T, t) \in \rho$ (i.e., $C(T, t)$ was already processed) or $\omega(S(T))=t$ (that is, $S(T)$ was processed when $CLOCK=t$)
NEXT	a function indicating the valid state transitions for a given input message and a given initial state
LOCK_TABLE	a function from π into $\tau \times M^* \cup \{\lambda\}$ where M is the set of all possible messages $LOCK_TABLE(j)=(T, Q)$ iff page j is locked for transaction T and Q is a sequence of messages waiting for T to unlock j (Q may be empty) $LOCK_TABLE(j)=\lambda$ iff page j is free

Note: if $v=(j, T) \in VERSION_POOL$, we use $v.PAGE$ and $v.TRANS$ to denote j and T , respectively.

We now introduce by definition a sequence of sets and functions that will abbreviate the description of the operations. We begin by defining the following subsets of $VERSION_POOL$:

- (1) $v \in UNCERTIFIED_POOL$ iff $v \in VERSION_POOL$ and $STATE(v.TRANS)=ACTIVE$
- (2) $v \in DISCARDED_POOL$ iff $v \in VERSION_POOL$ and $STATE(v.TRANS)=ABORTED$
- (3) $v \in CERTIFIED_POOL$ iff $v \in VERSION_POOL$ and $STATE(v.TRANS)=COMMITTED$
- (4) $v \in SAVED_POOL$ iff $v \in VERSION_POOL$ and $STATE(v.TRANS)=COMMITTING$

We proceed by introducing two functions, $LAST_STATE$ and $ACTIVE_STATE$, describing database states. $LAST_STATE$ is a function from pages into transactions giving, for each page, the last committed transaction, if any, that wrote on that page:

- (5) $LAST_STATE: \pi \rightarrow \tau \cup \{\lambda\}$ such that
 $LAST_STATE(j) = T$ iff $(j, T) \in CERTIFIED_POOL$ and $\nexists T' ((j, T') \in CERTIFIED_POOL \Rightarrow CTIME(T') < CTIME(T))$

$LAST_STATE(j) = \lambda$ iff $\nexists T' ((j, T') \in CERTIFIED_POOL)$

Note that, in our abstract discussion, it suffices to indicate in $LAST_STATE(j)$ the transaction that created the last certified version, since versions

are just pairs (j, T) .

ACTIVE STATE is a function mapping a page j and a timestamp t into the last committed or committing transaction T that wrote on j before t , if any:

(6) ACTIVE_STATE: $\pi \times N \rightarrow \tau \cup \{\lambda\}$ such that

ACTIVE_STATE(j, t) = T iff
 $(j, T) \in \text{CERTIFIED_POOL} \cup \text{SAVED_POOL}$ and
 $\text{CTIME}(T) < t$ and
 $\nexists T' ((j, T') \in \text{CERTIFIED_POOL} \cup \text{SAVED_POOL}$ and
 $\text{CTIME}(T') < t \Rightarrow \text{CTIME}(T') < \text{CTIME}(T)$)

ACTIVE_STATE(j, t) = λ iff
 $\nexists T ((j, T) \in \text{CERTIFIED_POOL} \cup \text{SAVED_POOL}$ and
 $\text{CTIME}(T) < t)$

It is also convenient to define two additional subsets of VERSION_POOL:

(7) $(j, T) \in \text{LAST CERTIFIED_POOL}$ iff
 $\text{LAST_STATE}(j) = T$

(8) $(j, T) \in \text{PERMISSIBLE READ POOL}$ iff
 $\exists T' (T' \in \tau$ and $\text{STATE}(T') = \text{ACTIVE}$ and
 $T'.\text{Type} = \text{Query}$ and
 $\text{ACTIVE_STATE}(j, \text{BTIME}(T')) = T)$

That is, (j, T) is a version potentially needed by some active query T' .

A high-level description of the message processing routines of the storage subsystem is given in Figure 1 at the end of this section. The code is self-explanatory and includes all routines, except those related to concurrency control. The correctness of these routines, both in a centralized and in a distributed DBMS, is left to Section 4. In particular, the role of the assignments in italics and the bracketed sequences of statements, which are not a proper part of the algorithm, is discussed in Section 4.

The recovery strategy adopted is simple and guarantees that VERSION_POOL always contains, for each page j , AT LEAST the version created by the last committed transaction that wrote on j , and the version (there will be just one), if any, created by a transaction that successfully completed the first phase of the two-phase commit protocol but which has not yet terminated. We consider that this is the minimum set of versions that must be guaranteed not to be lost by the storage subsystem.

A garbage collection routine is also shown in Figure 1 for the sake of completeness, but it is left unspecified when it should be called.

The concurrency control routines implement a mixed strategy, as in [CFLNR]. Update transactions follow a two-phase locking policy, implemented by the routines SYNC_BEFORE and SYNC_AFTER, where locks are set for each logical page before it is read or written (by an update transaction), and released at commit time. Query transactions are synchronized with update transactions by a version selection policy, implemented by the READ operation. Given a read message $R(j, Q, k)$ of a query transaction Q with begin timestamp t , the READ operation selects the most recent version of page j in CERTIFIED_POOL or SAVED_POOL created before t . If no version of page

j , created before t , exists in CERTIFIED_POOL or in SAVED_POOL, then Q is aborted and started again. A version v in SAVED_POOL, created by a committing transaction T , is selected if t is greater than the clock value at the time $S(T)$ was processed, and $R(j, Q, k)$ is processed before T commits. In this case $R(j, Q, k)$ will be enqueued waiting for T to commit and unlock page j (LOCK_TABLE, a data structure not discussed due to space limitations, is used to enqueue messages). If T aborts, the selection process starts all over again.

FIGURE 1

High-Level Description of Message Processing Routines

```
MSG_PROCESSOR(M;v,MA,RC)
/*
  input : M - message to be processed
  output: v - version read, if any
  MA - list of messages that become
        unblocked
  RC - return code, with the following
        values:
        0, if M was successfully processed
        1, if the transaction aborted
        2, if processing of M was blocked
*/
begin /* test if message can be accepted */
  TEST_INPUT(M;RC);
  /* synchronize before processing */
  SYNC_BEFORE(M,RC;RC);
  /* process message */
  PROC(M,RC;v,RC);
  /* abort if something went wrong */
  if RC = 1 then ABORT(T);
  /* synchronize after processing */
  SYNC_AFTER(M,RC,v;MA,RC);
end

TEST_INPUT(M;RC)
begin /*
  signal to abort if invalid input
  (M.TRANS denotes the transaction that
  issued M)
*/
  if M  $\neq$  RE and NEXT(M,STATE(M.TRANS)) =  $\lambda$ 
  then RC := 1
  else RC := 0
end

PROC(M,RC;v)
begin
  if RC  $\neq$  0 then return;
  case M of
    B(T,t)   $\rightarrow$  BEGIN(T,t)
    R(j,T,k)  $\rightarrow$  READ(j,M;RC,v)
    W(j,T,k)  $\rightarrow$  WRITE(j,T)
    S(T)     $\rightarrow$  SAVE(T)
    C(T,t)   $\rightarrow$  COMMIT(T,t)
    A(T)     $\rightarrow$  ABORT(T)
    RE       $\rightarrow$  RESTORE
  endcase
end

BEGIN(T,t)
begin
  BTIME(T) := t;
  [ STATE(T) := ACTIVE;  $\rho := \rho \cup \{B(T,t)\}$  ]
end
```

```

WRITE(j,T)
begin
  [ VERSION_POOL := VERSION_POOL  $\cup$  {(j,T)};
     $\rho := \rho \cup \{W(j,T)\}$  ]
end

SAVE(T)
begin
  CTIME(T) := CLOCK;
  [ STATE(T) := COMMITTING;  $\rho := \rho \cup \{S(T)\}$  ]
end

COMMIT(T,t)
begin
  CTIME(T) := t;
  [ STATE(T) := COMMITTED;  $\rho := \rho \cup \{C(T,t)\}$  ]
end

ABORT(T)
begin /* change state before dropping versions */
  [ STATE(T) := ABORTED;  $\rho := \rho \cup \{A(T)\}$ ;
    VERSION_POOL := VERSION_POOL -
      {v $\in$ VERSION_POOL/v.TRANS = T} ]
end

RESTORE
begin /*
  abort active transactions and
  THEN drop their versions
  */
  for each transaction T do
    if STATE(T) = ACTIVE
      then [ STATE(T) := ABORTED;
             $\rho := \rho \cup \{A(T)\}$ ; ]
    VERSION_POOL := LAST_CERTIFIED_POOL  $\cup$  SAVED_POOL;
  /* reconstruction of LOCK_TABLE */
  for each page j do
    if there is T such that (j,T) $\in$ SAVED_POOL
      then LOCK_TABLE(j) := (T, $\lambda$ );
      else LOCK_TABLE(j) :=  $\lambda$ 
    end
  end
end

READ(j,M;RC,v)
begin /*
  select version to be read based on
  transaction type
  */
  T := M.TRANS;
  if T.type = Update
    then if (j,T)  $\in$  VERSION_POOL
          then /* select version created by T */
              v := (j,T);
            else /* select last certified ver. */
              v := (j, LAST_STATE(j));
            else /*
              select last certified or saved
              version created before the query
              began
              */
              begin T' := ACTIVE_STATE(j,BTIME(T));
                 case T' =  $\lambda$ 
                    $\rightarrow$  v := (j, $\lambda$ ); RC := 1;
                   STATE(T')  $\neq$  COMMITTING
                    $\rightarrow$  v := (j,T);
                   else begin RC := 2;
                        let LOCK_TABLE(j)=(T',Q);
                        LOCK_TABLE :=
                          (T', Q  $\cup$  {M});
                        end
                   endcase
              end
            end
  end
end

```

```

GARBAGE_COLLECT:
begin
  VERSION_POOL := UNCERTIFIED_POOL
     $\cup$  SAVED_POOL
     $\cup$  LAST_CERTIFIED_POOL
     $\cup$  PERMISSIBLE_READ_POOL
end

```

4. HIGH_LEVEL CORRECTNESS ISSUES

4.1 Reliability Correctness

4.1.1 Reliability Correctness Criterion

Recall that ρ is the set of messages processed thus far by the storage subsystem. Independently of the data structures, we define the following set of versions:

- D1. $(j,T) \in$ saved-pool iff
 $W(j,T) \in \rho$ and $S(T) \in \rho$ and
 $\neg(\exists t(C(T,t) \in \rho) \text{ or } A(T) \in \rho)$
- D2. $(j,T) \in$ last-certified-pool iff
 $W(j,T) \in \rho$ and
 $\exists t(C(T,t) \in \rho$ and
 $\forall T' \forall t'(W(j,T') \in \rho \text{ and } C(T',t') \in \rho \Rightarrow t' \leq t)$)

That is, $(j,T) \in$ saved-pool iff (j,T) was created by a committing transaction T and $(j,T) \in$ last-certified-pool iff (j,T) was created by a committed transaction T which was the last committed transaction to create a version of j.

Note: all sets of versions defined directly in terms of ρ will be underscored; subsets of VERSION_POOL will be denoted in capital letters.

Now, our reliability correctness criterion goes as follows:

Reliability Correctness Criterion

Assertion A1 must be an invariant of the storage subsystem:

- A1. saved-pool \cup last-certified-pool \subseteq VERSION_POOL

We use the term "invariant" with the usual sense, that is, an assertion A is an invariant of an algorithm r iff, for any computation $I=(I_0, I_1, \dots)$ of r, if A is true in I_0 then A is true in every other state I_j of I. The reliability correctness criterion is justified on the grounds that, if it holds for a particular implementation r of the storage subsystem, then if any computation of r is stopped in any state I_j , we know that the value of VERSION_POOL (or of a representation of VERSION_POOL) in I_j contains all last certified and all saved versions created thus far. Hence, the database can be restored to contain all last certified versions and all versions created by transactions that passed the first stage of the two-phase commit, but which have not yet terminated. We consider that this is the minimum set of versions that must never be lost by any correct storage subsystem.

The concept that A is an invariant of an algorithm r, however, depends on specifying which operations are considered atomic. We approach this problem by adopting an incremental correctness strategy in the spirit of the description of the storage subsystem. We first assume that all tests and assignments are atomic and prove that assertion A1 is an invariant of the high-level algorithm. The choice of tests and assignments as atomic statements is justified since these are the basic statements on which if-then-elses, loops and case statements are built. Then, for a particular implementation of the storage subsystem, we prove (hopefully by modifying the first proof) that A1 (or its translation in terms of the data structures of the implementation) is an invariant of the algorithm describing the implementation. The next subsection addresses the first step.

4.1.2 Proving Reliability Correctness of the High-level Algorithm

As previously discussed, the first step towards establishing the correctness of the storage subsystem is to show that assertion A1 is an invariant under the tests and assignments of the high-level description. The result follows trivially for tests since they do not affect the value of any data structure. As for assignments, the correctness proof is slightly laborious, but not difficult, and follows using standard techniques. We summarize below the major points of the proof (see full paper for the details).

We first observe that the set ρ of messages processed thus far by the storage subsystem must be treated as an (auxiliary) proof variable [Ow], as mentioned in Section 3, that is, as a variable that does not affect the computation of the algorithm, but which is useful for proof purposes. This proof variable is updated by auxiliary assignments in BEGIN, SAVE, COMMIT and ABORT which, we stress, are not a proper part of the algorithm. To simplify the definition of certain assertions, we also added an auxiliary assignment to RESTORE that simulates the abortion of active transactions at the time RESTORE is executed. All auxiliary assignments are printed in italics in Figure 1. For the purposes of the proof, in each of these routines, we consider that the auxiliary assignment and a certain true assignment form a single atomic action (which is indicated by enclosing both assignments within brackets in each routine).

The second observation is typical of correctness proofs that rely on invariants. As is often the case, it is difficult to directly prove that the original assertion is invariant, so one proves instead that a stronger assertion is an invariant. Thus, we first strengthened assertion A1 to be:

A2. saved-pool \cup last-certified-pool \cup
uncertified-pool \subseteq VERSION POOL

where saved-pool and last-certified-pool were defined in Section 4.1.1 and uncertified-pool is defined as follows:

D3. $(j, T) \in$ uncertified-pool iff
 $W(j, T) \in \rho$ and
 $\neg(\exists t'(C(T, t') \in \rho) \text{ or } A(T) \in \rho \text{ or } S(T) \in \rho)$

This was reasonable because, intuitively, the algorithm "moves" new versions from uncertified-pool to saved-pool as transactions complete the first phase of the two-phase commit protocol, and from saved-pool to last-certified-pool as transactions commit. Our problem now is to show that A2 is an invariant (this almost trivializes the proof - see the last remark of this subsection).

It is still hard to prove that A2 is an invariant. Intuitively, the three new sets introduced, saved-pool, last-certified-pool and uncertified-pool, are defined in terms of the existence of certain messages in ρ and of the time transactions committed. This information is indeed available to the algorithm, but in the form of the functions STATE and CTIME, which were introduced exactly for this purpose. So, we first proved that certain auxiliary assertions describing properties of STATE and CTIME were invariant (not shown to save space).

From the invariance of these assertions, it follows that A3 is also an invariant:

A3. saved-pool \cup last-certified-pool \cup
uncertified-pool \subseteq resilient-pool

where resilient-pool is defined as follows:

D4. $(i, U) \in$ resilient-pool iff
 $W(i, U) \in \rho$ and
(STATE(U) = ACTIVE or
STATE(U) = COMMITTING or
STATE(U) = COMMITTED and
 $\forall U'(STATE(U') = COMMITTED \text{ and } W(i, U') \in \rho \Rightarrow$
CTIME(U') \leq CTIME(U)))

Now, consider the following assertion:

A4. resilient-pool \subseteq VERSION POOL

We can also show that A4 is an invariant. Finally, since A3 and A4 are invariants and since they trivially imply A2, then A2 is also an invariant, as was to be shown.

To conclude, at this level of abstraction, the reliability correctness proof amounts to showing that, if any execution is stopped at any point, no critical version is lost, on the assumption that all data structures are non-volatile (i.e., survive a memory crash), and that very-high level assignments are atomic actions. In another perspective, the proof checks if the order in which the various data structures are updated is correct, which is one of the difficulties one must face when designing reliability strategies. In fact, the proof led us to detect that the state of the transaction had to be changed in ABORT and RESTORE before updating VERSION POOL, as otherwise A2 would not be an invariant.

4.2 Concurrency Correctness

4.2.1 Model of a High-Level Execution

Let us assume that the DBMS is centralized so that there is just one storage subsystem. We model an execution of the high-level algorithm by a multi-version log (MV log) as in [BG1], except that the reads-from relation is present in the definition.

Briefly, an MV log for a set T of transactions is a triple $L=(S,<,h)$, where S is a set of operations (BEGIN, READ, WRITE, etc.) executed on behalf of the transactions by the storage subsystem, $<$ is a partial order on S such that $O < O'$ iff O completed execution before O' began executing (it is not necessary to assume that the implementation in question processes operations sequentially at this point) and h is a function mapping each read operation into the write operation that created the version selected. There are restrictions on the pair $(S,<)$ to capture operation sequencing, which we omit for reasons of brevity.

Usually, an MV log L is considered "correct" iff L is 1-serializable [BGI], that is, iff L is equivalent to a serial log where there is always just one version of each page. However, when recovery issues are also at stake, 1-serializability has to be qualified. To begin with, only versions created by committed transactions matter, so we must project logs onto the operations executed on behalf of committed transactions before testing for 1-serializability. We then say that a log is C-serializable iff its projection on the set of committed transactions is 1-serializable.

But recovery considerations raise even more important questions. First of all, recall that in Section 4.1. we require that all versions in last-certified-pool and in saved-pool never be lost. Those in saved-pool cannot be lost to satisfy the requirements of the two-phase commit protocol. But when we decided to guarantee that, for each page j , the storage subsystem must never lose the version of page j created by the last transaction that wrote on j and committed, we implicitly assumed that these versions form the "true" state of the database. Therefore, we require that any log L be strongly C-serializable, that is, C-serializable and, moreover, T is the last committed transaction to write on page j in L iff T does so in the equivalent serial log (to preserve the "last" version).

Finally, since we cannot know when a failure will stop the computation, we have to guarantee that strong C-serializability is an invariant of the storage subsystem. Equivalently, we require that every prefix of a log be strongly C-serializable. A log satisfying this property is called continuously C-serializable [Ca].

To summarize, we have:

Concurrency Correctness Criterion

Any log modelling an execution of the storage subsystem must be continuously C-serializable.

4.2.2 Concurrency Correctness of the High-Level Algorithm

As in the reliability proof, we will first prove the correctness of the high-level algorithm, leaving to a later section implementation considerations.

Assume again a centralized DBMS. The concurrency correctness proof of the high-level algorithm is fairly straightforward and follows from standard results. We first observe that update transactions

obey the standard two-phase locking strategy with lock point at commit time (when all pages are unlocked) and do not make use of multiple versions. Therefore, all update transactions are serializable in commit order and, moreover, any prefix of a log, when projected on the transactions already committed, is also serializable in commit order.

Now, by definition of the read operation, a query with begin timestamp t reads the last certified version of each page created before t . When the version is not kept anymore, the query is aborted, and when the version is still in certification, the read is blocked waiting for the version to be certified (if the version is never certified because its creating transaction is aborted, the version selection process starts all over again). Therefore we may prove that a query is serialized after all update transactions that committed before t and before all update transactions that committed after t .

These remarks can be formalized into a proof that any log modelling an execution of the storage subsystem is continuously C-serializable (see full paper).

4.2.3 Remarks about a Distributed Environment

When we consider a DDBMS, there is a storage subsystem per node so that a global execution can be modelled by an indexed set $\{L_1, \dots, L_n\}$, where L_i is the log modelling the local execution of the storage subsystem at node i . Our previous discussion can be generalized to this scenario by assuming that each L_i is continuously C-serializable and that committed transactions are serialized in the same order in different nodes.

The only crucial correctness problem in a distributed environment concerns the synchronization of queries and updates since it depends on the generation of begin and commit timestamps. Indeed, to correctly synchronize queries, the commit timestamp generated for T has to be less than that generated for T' , if T commits before T' and T conflicts with T' in some node k . This follows because, in this case, the two-phase locking strategy (with lock point at commit time) forces T to be serializable before T' . The difficulty lies in that T and T' may conflict at a node k and be controlled by TMs at nodes n and n' (that is, the commit timestamp for T and T' may be generated at two potentially distinct nodes and may depend on a conflict at a third node). Furthermore, all commit messages sent on behalf of T have to carry the same commit timestamp.

The first step towards solving this problem is implementing a Lamport Clock [BGI] as follows. A local clock is maintained in each node; messages are timestamped with the value of the local clock when they are sent; when a message is received with timestamp t , if t is greater than the value of the local clock, the local clock is advanced to t .

The generation of begin timestamps presents no problems since it can rely only on the local clock of the TM governing the execution of the transaction. But the generation of commit timestamps is somewhat more complex. When an update transaction T termi-

nates, the TM governing its execution initiates the two-phase commit protocol. Save messages are sent to all sites where T made a write. All sites reply with Save Acks, denoted $AS_k(T)$ for each node k, carrying YES or NO. If all sites replied with YES, a timestamp $ts(T)$ is generated for T as follows:

$$ts(T) = \text{MAX}\{w(AS_k(T) / k \text{ is a site where } T \text{ made a write})\}$$

where $w(M)$ is the timestamp of message M. Commit messages $C_k(T, t)$ are then sent to each site k updated by T_k , where $t = ts(T)$; otherwise, Abort messages are sent.

We now indicate that this strategy of generating timestamps is correct, provided that no node processes two messages without advancing its clock. If T and T' are updates, we write $T \rightarrow T'$ iff there is a node k and a page j of k such that either T writes on j before T' writes on j, or T reads j before T' writes on j, or T writes on j before T' reads j. Since updates follow the two-phase locking strategy, we can prove the following:

PROPOSITION 1:

- (a) Let T and T' be two committed transactions. If $T \rightarrow T'$ then the commit message of T is processed before the save message of T' in some site k.
- (b) the relation \rightarrow , when restricted to committed or committing transactions, is acyclic.

We say that a commit timestamp strategy is correct iff, for any two committed transactions T and T', if $T \rightarrow T'$ then the commit timestamp of T must be less than the commit timestamp of T'. Note that, by Proposition 1(b), the correctness criterion is well defined. We can show that our commit timestamp function, $ts(T)$, meets the above criterion.

LEMMA 1: Let T and T' be two committed transactions. If $T \rightarrow T'$ then $ts(T) < ts(T')$.

This concludes our remarks about a distributed environment.

5. RELIABILITY CORRECTNESS OF AN IMPLEMENTATION

This section refines the high level routines and the abstract data types defined before, bringing them closer to a real implementation. As already mentioned, at least another round of refinement is necessary in order to complete the specifications since at this level we will not address all issues of efficiency and, therefore, will still abstract some operations.

We will consider essentially two types of failures: memory failures and I/O failures. Recall from Section 4.1 that a computation of a storage subsystem r was captured as a possible infinite sequence of states $I = (I_0, I_1, \dots)$. A state I_j of r can be understood as an assignment of values to the data structures of r. The data structures

used in the implementation will, as usual, be classified in:

- stable data structures: remain unchanged after a primary failure, i.e. a failure that affects the main memory;
- volatile data structures: their values are lost after a primary failure.

5.2 Refinement of the High Level Data Structures

Section 3 defined all abstract data structures that were later manipulated by the storage subsystem routines. Note that, at that moment, we had no notion of what a failure was and, consequently, we did not divide the structures into stable or volatile accordingly to whether they survived or not a system crash. In particular, the RESTORE routine makes use of the function STATE in order to recover the system. This would be impossible unless STATE survives system crashes. The task of this section can be pictured as follows: define convenient stable data structures in such a way that all pertinent information can be computed from them, at any moment.

To see what information need to be stable, return to the code of RESTORE on Section 3. It is clear that STATE can be computed from ρ . We also need to be able to compute SAVED POOL and LAST CERTIFIED POOL. Hence, we need CTIME and VERSION POOL. But CTIME also follows from ρ . We claim that VERSION POOL can also be obtained from ρ . Note that ρ contains a complete list of all the write messages that were processed by the system and these are the only ones (except for RESTORE itself) that alter the contents of VERSION POOL. In conclusion, if we make ρ into a stable program entity we can recompute all needed information after a crash. We shall shortly capture ρ into a log.

In fact, we claim that all that needs to be maintained by the system at any moment is ρ . Clearly, we can recompute the value of any other data structure at any instant by reading ρ over and over. At the time of a crash, it doesn't matter too much if recomputing the value of other data structures is inefficient. At normal operation, however, we cannot accept such a gross inefficiency. Therefore we define, in addition to the stable data structures convenient volatile data structures in such a way that, at normal operation, they can be easily maintained and can be efficiently combined to give all information needed by the system at any moment. Of course we will incur in the extra burden of reconstructing these volatile data structures at the time of (supposedly) rare crashes. All program variables defined in Section 3 will be kept as volatile data. In order to reflect the particularities of the implementation some of these data structures shall have their domain redefined.

To make the implementation more realistic, we shall also take into account the value of each page of the database. Clearly, we have to keep some values in a stable data structure. We shall define the

stable data structure MEM to hold these values. To speed up the computations involving MEM, another volatile data structure, MAP, shall also be used.

In a sense, the stable structures provide a trustworthy repository of information whereas the volatile structures speed up the computation. There is one additional complicator in the fact that, in practice, we can not manipulate data directly in secondary storage. It must be read into the main memory page buffers, manipulated and written back to secondary storage. Therefore, at any moment, part of our stable structures will be resident in volatile buffers which must be assumed to be lost following a crash. We have to insure that this loss of information will not cause the system to recover to an inconsistent state.

Remarks:

- strictly speaking, CTIME is not always obtainable from ρ following a crash, since we can not compute CTIME(T) if STATE(T)=COMMITTING at the time of the crash. In this case the value of CTIME(T) will be lost.
- during the recovery process, we need the value of CTIME(T) only for those T such that STATE(T) = COMMITTED when the system crashes. This information is all that is needed in order to compute LAST STATE and hence LAST CERTIFIED POOL. But in this case we can use the fact that CTIME(T) = t iff C(T,t) is in ρ .
- after the recovery is complete and when the system resumes normal operation again, the READ routine is the only one that will make use of CTIME not only for those transactions T that were committed at the time of failure but also for those T that were then committing. This need is clear since READ uses the ACTIVE STATE function. But we cannot recover CTIME for the committing transactions.
- an alternative would be to manipulate save messages in the form S(T,t), exactly as was done with the C(T,t) messages. In this way we could recuperate CTIME also for those transactions T that were committing at the time of crash. The save timestamp now present in messages of the form S(T,t) is obtained simply by taking the value of the local clock when S(T,t) is executed. We already had CTIME(T)=CLOCK in SAVE(T).

We now give the definitions of each data structure:

STRUCTURE	INTERPRETATION
MEM	a stable sequence of N sectors. Each sector can hold the value of exactly one page
MAP	a volatile sequence of N bits. Indicates which sectors of MEM are free: for each i in [1,M], MAP(i)=0 iff the i th sector of MEM is free, otherwise, MAP(i)=1.
LOG_BUF	a volatile sequence of elements from $M \cup [1,M] \times \tau \times [1,N]$. A LOG_BUF

record can hold either a message or a triple of the form (j,T,m). The triples are written in LOG_BUF by the WRITE routine. Hence, (j,T,m) in LOG_BUF indicates that transaction T wrote the value of a version for page j on memory sector m. (The same applies for STAB_LOG records).

STAB_LOG	a stable sequence of elements from $M \cup [1,M] \times \tau \times [1,N]$.
MEM_BUF	a volatile subset of $([1,N] \cup \{\lambda\}) \times VAL$. Here, VAL represents the domain of values for the pages. It will be left unspecified since its specific form is unimportant. Each element of MEM_BUF is a pair (k,v) where k indicates a sector number in MEM and v is the contents of that sector.
VERSION_POOL	a volatile subset of $[1,M] \times \tau \times [1,N]$. An element of VERSION POOL in the form (j,T,m) indicates that transaction T has created a version for page j and sector m of MEM holds the value of that version.

All the other functions mentioned in Section 3 are kept as volatile structures and are defined exactly as before.

Remarks:

- it is not strictly necessary for STAB_LOG and LOG_BUF to have a sequential nature. They could as well be defined as sets. All the timing information that is needed is given by the functions BTIME and CTIME. During normal operation they are readily available. During recovery time, BTIME can be obtained from the B(T,t) messages and CTIME from the C(T,t) and S(T,t) messages, regardless of their particular orderings in the logs.
- if STAB_LOG is taken as a set, however, we need to reprogram RESTORE as given below since at that point we read the log backwards, an operation that presumes a time ordering of the log records.

5.3 Rewriting the Storage Subsystem Routines

The present implementation requires a few new routines in order to manipulate the buffers. In addition we will list below only the routines that have to be reprogrammed. More specifically, the routines:

1. MSG_PROCESSOR, TEST_INPUT and PROC need not be changed since they merely govern the overall flux of the algorithm;
2. SYNC_BEFORE, SYNC_AFTER, LOCK, UNLOCK and UNLOCK_ALL also need not be changed since they are needed solely for concurrency control purposes. They never manipulate the new data structures or the ones that have been redefined;
3. READ needs slight changes so that it now returns

- the value of the page requested;
4. BEGIN, SAVE, COMMIT, ABORT and WRITE will have to be extended in order to manipulate the new structures;
 5. RESTORE will have to be substantially rewritten.

The new routines are:

FLUSH_LOG transfers all records in the log buffer to the log in secondary storage.

FLUSH_MEM transfers the contents of the page buffer to secondary storage.

SELECT_MAP finds a free position in the MAP vector. Since this is an easy task we will leave SELECT_MAP unspecified.

READBACK reads STAB_LOG backwards returning one record per call. This is only a matter of proper pointer management and will also be omitted.

Note how close the routines are to their abstract counterparts (except for RESTORE) defined in Section 3. Based on the proof of correctness of the high level algorithm developed in Section 4 we can construct a proof of correctness for this implementation. We need additional assertives that will explicitate the behaviour of MEM, MAP, MEM_BUF, STAB_LOG and LOG_BUF. From these, a new reliability correctness criterion can be stated. Although not hard, the proof is a bit involved and laborious. For the sake of brevity, we will not detail the proof in this abstract. A more extensive development is carried out in the full paper.

The code for the routines mentioned in (3)-(5) above, as well as the code for the new routines, is displayed in Figure 2 below. We use the following conventions:

- a) // indicates the concatenation operator
- b) for a string x and a non-negative integer k , the prefix of length k of x is denoted by $k \setminus x$. The suffix of length k of x is written $x \setminus k$.

FIGURE 2

```

FLUSH_LOG
begin /* one record at a time */
  while |LOG_BUF| > 0 do
    begin
      STAB_LOG := STAB_LOG // 1 \ LOG_BUF;
      LOG_BUF := LOG_BUF \ (|LOG_BUF| - 1)
    end
  end
end

FLUSH_MEM
begin /* one record at a time */
  for all (k,v) ∈ MEM_BUF do MEM(k) := v;
  MEM_BUF := ∅
end

```

```

BEGIN(T,t)
begin
  BTIME(T) := t;
  STATE(T) := ACTIVE;
  LOG_BUF := LOG_BUF // B(T,t);
  FLUSH_LOG;
end

WRITE(j,T,v)
/*
  v represents the value of the page to
  be written
*/
begin
  if ∃s ((j,T,s) ∈ VERSION_POOL)
  then k := s;
  else begin
    k := SELECT_MAP;
    MAP(k) := 1;
    VERSION_POOL := VERSION_POOL ∪
      {(j,T,k)}
  end
  /*
    q contains a copy of sector k, if any,
    that is in MEM_BUF
  */
  q := {(k,u) / (k,u) ∈ MEM_BUF for some u};
  MEM_BUF := MEM_BUF - q ∪ {(k,v)};
  LOG_BUF := LOG_BUF // (j,T,k);
end

SAVE(T,t)
begin
  CTIME(T) := CLOCK;
  STATE(T) := COMMITTING;
  FLUSH_MEM;
  LOG_BUF := LOG_BUF // S(T,t);
  FLUSH_LOG;
end

COMMIT(T,t)
begin
  CTIME(T) := t;
  STATE(T) := COMMITTED;
  LOG_BUF := LOG_BUF // C(T,t);
  FLUSH_LOG;
end

ABORT(T)
begin
  /* change state before dropping versions */
  STATE(T) := ABORTED;
  q := {(j,T,s) / (j,T,s) ∈ VERSION_POOL
    for some j and some s};
  VERSION_POOL := VERSION_POOL - q;
  LOG_BUF := LOG_BUF // A(T);
  FLUSH_LOG;
end

```

```

RESTORE
begin /* initialize */
  for all j in [1,N] do
    begin
      MAP(j) := 0;
      LAST_STATE(j) := λ;
      LOCK_TABLE(j) := λ;
    end;
  VERSION_POOL := ∅;
  LOG_BUF := nil;
  MEM_BUF := ∅;
  /* read the log backwards and reconstruct STATE, CTIME, VERSION_POOL and MAP */
  repeat
    r := READBACK;
    case r of
      B(T,t) → if ¬(STATE(T) ∈ {COMMITTED, COMMITTING, ABORTED})
                then STATE(T) := ABORTED
      C(T,t) → begin
                  STATE(T) := COMMITTED;
                  CTIME(T) := t;
                end
      S(T,t) → if ¬(STATE(T) ∈ {ABORTED, COMMITTED})
                then begin
                      STATE(T) := COMMITTING;
                      CTIME(T) := t;
                    end
      A(T)    → STATE(T) := ABORTED
      (j,T,m) → case STATE(T) of
                  COMMITTING →
                    begin
                      VERSION_POOL := VERSION_POOL ∪ {(j,T,m)};
                      MAP(m) := 1;
                    end
                  COMMITTED →
                    begin
                      q := {(j,T',m') ∈ VERSION_POOL / STATE(T') := COMMITTED, some m'};
                      if q = ∅
                        then
                          begin
                            VERSION_POOL := VERSION_POOL ∪ {(j,T,m)};
                            MAP(m) := 1;
                          end
                        else
                          begin
                            let q := (j,T',m');
                            if CTIME(T') < CTIME(T)
                              then
                                begin
                                  MAP(m) := 1;
                                  MAP(m') := 0;
                                  VERSION_POOL := VERSION_POOL - {(j,T',m')} ∪ {(j,T,m)};
                                end
                              end
                        end
                    end
                endcase
    endcase
  until end-of-log;
  /* reconstruct LAST_STATE and LOCK TABLE */
  for all (j,T,m) in VERSION_POOL do
    case STATE(T) of
      COMMITTING → LOCK_TABLE(j) := (T,∅)
      COMMITTED  → LAST_STATE(j) := T
    endcase
  end.

```

```

READ(j,M;RC,v)
begin T := M.TRANS;
  if T.type = UPDATE
  then if  $\exists m ((j,T,m) \in \text{VERSION\_POOL})$ 
  then /* select version created by T */
    v := MEM(m);
  else /* select last (free) certified version if any */
    begin
      T' := LAST_STATE(j);
      if T' =  $\lambda$ 
      then begin RC := 1; v :=  $\lambda$  end
      else begin
          let (j,T',m)  $\in$  VERSION_POOL;
          v := MEM(m)
        end
      end
    end
  else /*
    select last certified or saved version
    created before the query began
  */
    begin T' := ACTIVE_STATE(j,BTIME(T));
      case T' =  $\lambda$ 
      → begin v:= $\lambda$  ; RC := 1; end
      STATE(T')  $\neq$  COMMITTING
      → begin
          let (j,T',m)  $\in$  VERSION_POOL;
          v := MEM(m)
        end
      else begin
          RC := 2;
          let LOCK_TABLE(j) = (T',Q);
          LOCK_TABLE(j) := (T', Q  $\cup$  {M})
        end
      endcase
    end
  end
end

```

6. CONCLUSIONS

The description of a local storage subsystem was analysed by adopting a multi-level, incremental design strategy. Initially a very high level description of the reliability and concurrency control strategies was given, which permitted a clear analysis of the interplay between concurrency control and reliability control, as well as checking if the order in which the various data structures are updated was correct. The second level of abstraction concentrated on the critical question of analysing the behavior of the strategies when some data structures are volatile and when I/O operations may fail. However, the subsystem was not completely specified in this extended abstract since the last refinement, having to do with efficient access to versions, was not addressed.

We believe that, without such design strategy, a correctness proof of the storage subsystem would not be possible. Or, at least, the strategy helped increase the degree of confidence in the correctness of the algorithm.

It should be noted that the storage subsystem was designed having in mind a distributed DBMS. As far as concurrency control goes, Section 4.2.3 discusses the only critical point besides global deadlock detection. However, the integration of the storage subsystem with other recovery layers of the DBMS, such as reintegration of a down site, must be addressed in more detail.

The high level algorithm presented in Section 3 is also a good vehicle to investigate the usual reliability strategies. By choosing different refinements, we believe that it is possible to analyse classical log recovery strategies, intention lists strategies, shadow page algorithms, and some of the newer multi-version integrated algorithms, similar to that described in Section 5.

REFERENCES

- [ABG] Attar, R., P.A. Bernstein, and N. Goodman, "Site initiation and back-up in a distributed database system", Proc. 6th Berkeley Workshop (Feb. 1982), 185-202.
- [BBGLS] Beeri, E., P.A. Bernstein, N. Goodman, M.Y. Lai, and D.E. Shasha, "A concurrency control theory for nested transactions", Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (Aug. 1983).
- [Be] Best, E. "Atomicity of activities", in Net Theory and Applications, Brauer, W. (ed.), Lecture Notes in Computer Science, Vol.84, Springer-Verlag (1979).
- [BG1] Bernstein, P.A., and N. Goodman, "Multi-version concurrency control - Theory and algorithms", ACM Trans. on Database Systems, Vol. 8, No 4 (Dec. 1983), 465-483.

- [BG2] Bernstein, P.A., and N. Goodman, "The failure and recovery problems for replicated databases", Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (Aug. 1983).
- [BCH] Bernstein, P.A., N. Goodman, and V. Hadzilacos, "Recovery algorithms for database systems", Proc. 1983 IFIP Conference, Paris (Sept. 1983).
- [BHR] Bayer, R., H. Heller, and A. Reiser, "Parallelism and recovery in database systems", ACM Trans. on Database Systems, Vol. 5, No 2 (June 1980).
- [Ca] Casanova, M.A., The Concurrency Control Problem for Database Systems, Lecture Notes in Computer Science, Vol. 116, Springer-Verlag (1981).
- [CFLNR] Chan A., S. Fox, T.A. Landers, A. Nori, and D. Ries. "The implementation of an integrated concurrency control and recovery scheme", Proc. ACM SIGMOD Conf. on Management of Data (June 1982), 184-191.
- [Gr] Gray, J.N., "Notes on database operating systems", in Operating Systems: an Advanced Course, R. Bayer, R.M. Graham and G. Seegmuller (eds.), Lecture Notes in Computer Science Vol.60, Springer-Verlag, New York, 391-481.
- [GMBL] Gray, J.N., P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo, and I. Traiger, "The recovery manager of the System R database manager", ACM Comp. Surveys, Vol. 13, No 2 (June 1981), 223-242.
- [GCDFRS] Goodman N., A. Chan, U. Dayal, S. Fox, D. Ries, and D. Skeen, "A recovery algorithm for a distributed database system", Proc. 2nd. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (Mar. 1983), 8-14.
- [Hal] Hadzilacos V., "An operational model for database system reliability", Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (Mar. 1983), 244-257.
- [Ha2] Hadzilacos V., "An algorithm for minimizing roll back cost", Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (Mar. 1982), 93-97.
- [HR] Harder, T. and A. Reuter, A Systematic Framework for the Description of Transaction Oriented Logging and Recovery Schemes, Tec. Rep., Univ. Kaiserslautern.
- [HS] Hammer, M., and D. Shipman, "Reliability mechanism for SDD-1: a system for distributed databases", ACM Trans. on Database Systems, Vol. 5, No 4 (Dec. 1980), 431-466.
- [Ke] Keller, R.M. "Formal verification of parallel programs", Comm. ACM, Vol. 19, No 7 (1976), 371-384.
- [Ko] Kohler, W.H. "A survey of techniques for synchronization and recovery in decentralized computer systems", Comp. Surveys, Vol. 13, No 2 (June 1981), 149-184.
- [La] Lamport, L. "Proving the correctness of multiprocess programs", IEEE Transactions on Software Eng., Vol. 3, No 2 (1977), 125-143.
- [Li] Lindsay, B.G. "Single and multi-site recovery facilities" in Distributed Data Bases: An Advanced Course, I.W. Draffan and F. Poole (eds.), Cambridge University Press, Cambridge 1980.
- [Lo] Lorie, R.A. "Physical integrity in a large segmented database", ACM Trans. on Data Base Systems, Vol. 2, No 1 (Mar. 1977), 91-104.
- [Ly] Lynch, N. "Multi-level atomicity - A new correctness criterion for database concurrency control", ACM Trans. on Data Base Systems, Vo. 8, No 4 (Dec. 1983), 484-502.
- [Ow] Owicki, S. Axiomatic Proof Techniques for Parallel Programs, TR-75-251, Dept. of Comp. Science, Cornell Univ. (July 1975).
- [PK] Papadimitriou, C.H., and Kanellakis, "On concurrency control by multiple versions", ACM Trans. on Database Systems, Vol. 9, No 1 (Mar. 1984), 89-99.
- [PPRS] Parker Jr., D.S., G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of mutual inconsistency in distributed systems", IEEE Trans. on Software Eng., Vol. SE-9, No 3 (May 1983), 240-237.
- [Re] Reed, D.P. "Implementing atomic actions on decentralized data", ACM Trans. on Computer Systems, Vol. 1, No 1 (Feb. 1983), 3-23.
- [RH] Reuter, A., and T. Harder, Principles of Transaction-Oriented Database Recovery, IBM Tech. Rep. RJ4214, San Jose Research Lab. (Mar. 1984).
- [SR] Stearns, R.E., and D.J. Rosenkrantz, "Distributed database concurrency control using before-values", Proc. ACM SIGMOD Conf. on Management of Data (April 1981), 74-83.
- [TGGL] Traiger, I.L., J. Gray, C.A. Galtieri, and B.G. Lindsay, "Transactions and consistency in distributed database systems, Vol. 7, No 3 (Sept. 1982), 323-342.
- [Ve] Verhofstad, J.M.S. "Recovery techniques for database systems", ACM Computer Surveys, Vol. 10, No 2 (June 1980), 167-196.