Model and Verification of a Data Manager Based on ARIES

DEAN KUO
CSIRO, Division of Information Technology

In this article, we model and verify a data manager whose algorithm is based on ARIES. The work uses the I/O automata method as the formal model and the definition of correctness is defined on the interface between the scheduler and the data manager.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Reasoning about Programs—*assertions, invariants, pre- and postconditions*; H.2.2 [**Data Management**]: Physical Design—*restart recovery*; H.2.4 [**Data Management**]: Systems—*transaction processing*

General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: ARIES, I/O automata, system failures

1. INTRODUCTION

For many applications, it is essential that a database management system (DBMS) maintains the data correctly, so that it reflects the effects of exactly the preceding committed transactions. One group of problems is caused by concurrent activities, and they are usually solved by the scheduler using a concurrency control algorithm. The other major source of problems is caused by the fact that the system maintains a buffer in memory (for efficient access), in which some of the data is cached. A system crash destroys the buffer, and so the database must be reconstructed during restart from values on disk (stable storage). Recovery algorithms usually rely on the log of system activities and are complex, therefore error prone, as the DBMS has no control as to when failures, such as crashes and

This work was done while the author was at the Basser Department of Computer Science, University of Sydney, Sydney, Australia and at the Distributed Systems Technology Center, University of Queensland, Brisbane. The work reported in this paper has been funded in part by the Cooperative Research Centers Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.
Authors' address: CSIRO Division of Information Technology, GPO Box 664, Canberra ACT 2601 Australia.

transaction rollbacks,[1] occur. The algorithm must ensure that it can recover from a failure at any time (even when it is recovering from a previous failure). To make recovery even more complex, the algorithm must also be efficient such that recovery time is kept to a minimum.

ARIES [Mohan et al. 1992] is a relatively new and important recovery algorithm that has been implemented in varying degrees in such systems as OS/2, DB2,[TM] Starburst, and Quicksilver. The key ideas of ARIES are that recovery repeats history (including actions from transactions that did not commit) and then undoes the actions of transactions that did not commit before the crash (these are called "loser transactions").

Formal methods have been advocated as a way to understand complex systems, as well as offering a framework in which to prove that the system, or some key algorithm, is correct. This has been done extensively for concurrency control algorithms [Bernstein et al. 1987; Fekete et al. 1990; Lynch et al. 1993], but little work has yet been done for recovery algorithms. Perhaps this is because recovery algorithms are so complex, but this makes them error prone and so makes careful analysis even more valuable. Our motivation is reflected in Selinger [1987] where she states,

> "one of the contributions that theory can make to systems is the confirmation that an algorithm is correct . . . Recovery—an example . . . of where more work is needed."

Verification of an algorithm requires a precise mathematical description of the algorithm so that we can state and prove its properties. Therefore one of the first steps in verification is to describe the algorithm in a precise language; that is, to model the algorithm in some formal method. This is useful in itself, even if verification is not done, because other papers that describe recovery algorithms use an imprecise combination of English and pseudocode. As a result, ambiguities may be revealed when a mathematical description is attempted.

This article presents a model and verification of a data manager whose algorithm is based on ARIES. The formal method used in I/O automata which is presented in Lynch et al. [1993], and Lynch and Tuttle [1987]. We have modeled ARIES very closely. In particular we include in our model: buffer management, write-ahead logging, asynchronous checkpointing, multiple data items per page, and the possibility of crashes occurring during restart processing. However, to keep the proof tractable, our model does not include features such as partial rollback or logical logging. We assume that there is a finite number of data items, each of which is stored in some page and never moves. We have followed the published account of ARIES [Mohan et al. 1992]; however, that paper is ambiguous with its description of checkpointing of the transaction table. Thus we had to make some choices that are discussed in more detail in Section 4.

[1]In Bernstein et al. [1987] and other literature, the term *abort* is used instead of *rollback*; we use the latter to keep our terminology consistent with ARIES [Mohan et al. 1992].
[TM]DB2, OS/2 are trademarks of the International Business Machine Corp.

There has been some related work in understanding recovery algorithms. In Bernstein et al. [1987], Gray and Reuter [1993], and Harder and Reuter [1983], algorithms are presented and classified but not verified. In Hadzilacos [1988], general conditions are proved on log maintenance, without which the information needed for recovery might not be available after a crash. However, that work does not show that specific recovery mechanisms use the information correctly. We have also studied other recovery algorithms with the approach taken here. In our first attempt [Kuo and Fekete 1992] a simple unrealistic data manager was modeled and verified. Checkpoints were not modeled, thus recovery had to scan the whole log. In Kuo and Fekete [1994], the *partial data item logging algorithm* from Bernstein et al. [1987] was modeled and verified. In that algorithm, restart requires two passes of the log. The first pass undoes the updates from loser transactions and the second pass is the redo pass where updates from committed transactions are redone. This is quite different from the method studied here. A preliminary version of this work appeared in Kuo [1992a].

The three main contributions of this paper occupy Sections 3–5. In Section 3, we present a specification of the data manager by first describing the interface between the data manager and the scheduler (the component in the DBMS that implements the concurrency control algorithm and then we define the correctness of the data manager in terms of the permitted sequences of interactions across this interface. This definition applies to a range of recovery algorithms, not only to ARIES. In Section 4, we present the precise model of the data manager whose recovery algorithm is based on ARIES; there is also an explanation of the methodology by which an informal account of an algorithm is converted into a tractable mathematical description. Then, in Section 5, we present the verification that shows that the data manager is correct. The verification proceeds through several intermediate results which make precise invariant relationships among the data structures used by the system as it executes. These relationships aid our understanding of the algorithm. The details of the proofs may be found in Kuo [1992b].

2. BACKGROUND

This section summarizes the necessary background knowledge; this covers the ARIES recovery algorithm and the I/O automaton formal method. Readers familiar with these topics may skip this section.

The main components for transaction processing in a DBMS are the transaction manager, the scheduler, and the data manager. This article focuses on the data manager and the interface between the data manager and the scheduler. We use this interface to define the correctness of the data manager and then prove that a particular algorithm (one that is based on a slightly simplified version of ARIES) is correct. In this section we give a brief description of the ARIES algorithm. In Section 4 of this article, we formally model the algorithm, giving a more detailed account. A comprehensive description can be found in Mohan et al. [1992].

We use the I/O automata formal method and an overview of it is presented in Section 2.2. A more extensive account is presented in Lynch et al. [1993] and Lynch and Tuttle [1987]. There are several reasons why we choose to use the I/O automata formal method; the main one is its support for abstraction—a central concern is to define correctness in terms of the interaction between a component and its environment. That is, the definition is independent of the data structures internal to the data manager and the internal processes. Also, there has been much work in specification and verification of concurrency control algorithms [Lynch et al. 1993] using this method. We hope in future to combine our current work with other results to show the correctness of the entire transaction processing subsystem in a DBMS.

2.1 ARIES

A recent recovery algorithm that enforces write-ahead logging (WAL) protocol is ARIES [Mohan et al. 1992]. It assumes that the scheduler provides strict executions [Bernstein et al. 1987]; that is, each read and write to a data item by a transaction is blocked until the last transaction that wrote to it has terminated. Checkpoints are taken asynchronously and are used to improve the performance of recovery.

During normal processing, the data manager processes transaction reads, writes, commits, and rollbacks. For reads and writes, the data manager fetches the data into the volatile cache if it is not already there. The operation (read or write) is then performed and recorded in the log if the operation is a write. Before the data can be flushed to the stable storage on disk, all the log records that recorded updates to the page must first be flushed to the stable log.[2] This restriction on when data items are flushed is called WAL. ARIES keeps track of the status of active transactions and dirty pages in the system using the transaction table and dirty pages table, respectively. Checkpoints are taken periodically; each takes a copy of these tables and writes them into an *end checkpoint* log record. A *begin checkpoint* log record is written to the log at the start of the checkpoint procedure.

After a crash, the effects of some updates from committed transactions may be missing from the stable database as the changed data values may not have been flushed to the disk; also the effects of other updates may be in the stable database even though the transaction had not committed before the crash. Thus some updates need to be redone while others require undo. ARIES' recovery algorithm requires three passes of the log to achieve recovery. The analysis pass reconstructs the transaction table and dirty pages table to reflect the state of the database. The redo pass re-installs all the missing updates by using the reconstructed dirty pages table to determine which updates require redo. An operation is redone by copying the after image of the operation back to the data item. Then the undo pass

[2]The term *stable log* is used to describe the part of the log that is stored in the stable storage.

restores old values for all the updates from transactions that did not commit before the crash by copying the before image of the operation to the data item. It uses the reconstructed transaction table to determine which updates require undo.

We have deliberately given a very brief and high-level description of ARIES as our model in Section 4 provides a comprehensive, formal and informal, description.

2.2 The I/O Automata Formal Method

The I/O automata method presented in Lynch et al. [1993] and Lynch and Tuttle [1987] is designed to model discrete event systems which contain components that operate concurrently. It has been extensively used to model and verify concurrency control algorithms [Fekete et al. 1990; Lynch et al. 1993], distributed algorithms [Lynch and Tuttle 1987], and network protocols [Fekete 1993]. The method is simple to use, readable, and it models abstraction cleanly—that is, we can define correctness independent of the implementation. This is essential if we are to understand the alternative algorithms that can be used for a component of a complex system.

In the I/O automata method, changes in the system being modeled are represented as actions. These are classified into three types: input, internal, and output actions. Input actions are generated by the environment. In our model of the data manager, the input actions are used to model the requests from the scheduler to the data manager. Internal and output actions are generated autonomously and output actions have the added property that they generate output to the environment. We use the internal actions to model the internal steps of the data manager and we use the output actions to model the acknowledgments of the completion of a request from the data manager to the scheduler. The set of all actions is referred to as *acts(A)* and the external actions, which are referred to as $ext(A)$, are the union of the input actions and the output actions. The internal and output actions are expressed by *preconditions* and *effects*. The *preconditions* describe the state the model must be in for the action to be enabled, and the *effects* show the transition from the old state to the new state. Input actions are always enabled so only the *effects* are given. Note that the actions of the I/O automaton are atomic—that is, there are no intermediate states showing only part of the effects of an action.

Each I/O automaton model has a set of states *states(A)*, and we usually describe the current state of the model by assigning a value to each of the variables in the model. We use the notation $S.Variable$ to refer to the state of the variable $Variable$ in state S. There is a nonempty subset, referred to as $start(A)$, of $states(A)$ which are the start states of the model. Each element in $start(A)$ represents a possible initial state for the model and models the initial state of the system (when the system is first installed).

The transition relationship in the I/O automaton is denoted as

$$steps(A) \subseteq (states(A) \times acts(A) \times states(A)).$$

Each element of this set is called a step of the automaton. An element $(s', \pi, s) \in steps(A)$ represents the model changing from the current state s' to the new state s via the action π. We say that the action π is enabled in state s' if $(s', \pi, s) \in steps(A)$. Input actions are always enabled, therefore if π is an input action and s' is a state, then there must exist s so that $(s', \pi, s) \in steps(A)$.

An *execution* β of the automaton A is an alternating sequence of states and actions; for example, $\beta = S_0\pi_1S_1\pi_2S_2, \ldots, S_{n-1}\pi_nS_n, \ldots$, where S_0 is a start state [i.e., $S_0 \in start(A)$], and for each i where $i > 0$, (S_{i-1}, π_i, S_i) is a step of A. From an execution, we can extract the *schedule* of the execution which is the subsequence of the execution that consists of actions only. For example, the schedule of the execution β is $sch(\beta) = \pi_1\pi_2\pi_3, \ldots, \pi_n, \ldots$.

The *behavior* of a schedule is the subsequence containing only the external (input and output) actions of the schedule. That is, the behavior is independent of state and of internal actions. Correctness is defined by stating the acceptable behaviors of the automaton model A and this is the approach that we have taken in our definition of correctness.

During recovery, the users are blocked from accessing the database. However, the I/O automata model is always input action enabled; that is, the I/O automata model can not block input actions from occurring. Therefore, in the model it is possible for the data manager to receive a read request from the scheduler during recovery. Instead of preventing the illegal inputs, the I/O automata method takes the approach that inputs are always enabled but the system may exhibit arbitrary behavior when unwanted inputs do occur. Thus our correctness definition will explicitly apply only to those behaviors where read and write requests do not occur at inappropriate times—that is, our correctness condition is of the form *if the input is correct, then the output is correct.*

In summary, the I/O automata method provides a way to precisely (mathematically) describe algorithms as well as statements of properties of the algorithm. Correctness is defined on the external behavior of the model and is independent of the state of the model or the internal actions. This provides a clean and abstract statement about the correctness of the model. It is then possible to use the model to verify that the algorithm that has been modeled is correct. We carry out this project for the ARIES algorithm in the rest of this article.

3. SPECIFICATION OF THE DATA MANAGER

The contribution of this section is a formal definition of the correctness of the data manager in a fashion independent of the internal data structures and internal processing. We can therefore apply this correctness definition to other protocols such as those described in Bernstein et al. [1987]. We have done this in recent work [Kuo and Fekete 1994]. However, ours is not the most general definition as it assumes that the scheduler provides strict schedules. Our correctness definition *can not* be used for data managers in

Table I. Input and Output Actions of the Data Manager

Input Actions	Output Actions
$ReqRead(T, x)$	$AckRead(T, x, v)$
$ReqWrite(T, x, v)$	$AckWrite(T, x, v)$
$ReqCommit(T)$	$AckCommit(T)$
$ReqRollbk(T)$	$AckRollbk(T)$
Crash	*AckRestart*

a DBMS that uses a nonstrict technique such as optimistic concurrency control.

In Section 3.1, we present the external (input and output) actions of the data manager and then in Section 3.2 we formally define correctness in terms of the sequence of these actions produced by the data manager.

3.1 External Actions of the Data Manager

The legal operations, during normal processing, that the scheduler can request to the data manager are read, write, commit, and rollback. Thus the input and output actions of the data manager during normal processing are the requests and acknowledgments of these operations. For example, we have the input action *ReqRead*(T, x) to model the request of a read by the transaction T to the data item x and the output action *AckRead*(T, x, v) to model the completion of the read with v as the value read.[3]

If the data manager fails to commit a transaction, it rolls back the transaction as if the scheduler requested a rollback. In our model, if the data manager successfully commits a transaction T, then the action *Ack-Commit*(T) occurs in the behavior; otherwise the action *AckRollbk*(T) occurs.

A system failure may occur at any time and is modeled by the input action *Crash*. Once a failure has occurred, recovery is enabled. The output action *AckRestart* signals the successful completion of recovery.

The input and output actions of the model are shown in Table I.

It is fundamental to the formal method that correctness must be defined in terms of sequences of these external actions. Thus instead of mentioning internal state and saying that a system is correct if the data values are appropriate, we instead say that it is correct if every read request receives the appropriate return value, reflecting the committed updates to that item. One subtle point should be mentioned. The acknowledgment of a request models the completion of a request. While the acknowledgment is outstanding, the scheduler does not know the status of the operation. For example, if the scheduler submitted a commit request but has not received the acknowledgment, then it does not know if the transaction has committed as the data manager may not have started processing the commit, or it may have processed the commit but has not sent the acknowledgment. In

[3]In our model, input and output actions are respectively prefixed by *Req* and *Ack* except for the action *Crash*.

recovery algorithms there is an atomic action that commits a transaction.[4] However, in our model, this step is not modeled as an external action, hence when a crash occurs during a commit, the behavior contains insufficient information to determine the status of the transaction. This, therefore, complicates our correctness definition. Our approach is to say that a sequence of external actions is correct if there exists a set of transactions that can be regarded as committed. This set must have certain properties such as being a subset of all the transactions that have requested a commit and their commit did not fail. Also, it must be a superset of all transactions that successfully committed. This gives a definition that deals only with externally visible activities, and hence applies to many different algorithms. In verifying a particular algorithm, there is no difficulty because we have access to the whole execution which includes the internal actions, and can thus pick the set of committed transactions exactly.

3.2 Correctness

In this section, we formally define correctness of a data manager. However, we first informally describe it. As noted, we require that each read by any transaction T return the appropriate value.

The appropriate value is informally defined as follows:

Case 1. If neither the transaction T nor any committed transaction have written to the data item then the appropriate value is the initial value, which we define as 0.

Case 2. If T itself wrote to the data item before the read, then the appropriate value is the value that T wrote.

Case 3. In the final case, some committed transactions have written to the data item but T has not. The appropriate value is then the value written by the *last* transaction that committed and wrote to the data item.

To shed light on why this is the intuitively correct definition, let us examine the behavior of the data manager. If neither system failures nor transaction rollbacks ever occur, then each read should return the value written by the transaction that last wrote to it, if such a transaction exists. If no transaction has ever written to it, then the read should return the initial value. Now suppose that failures do occur. Transactions are atomic; that is, if the transaction commits, then all its effects are visible, whereas if it aborts then none are. Also, to avoid such phenomena as cascading rollbacks [Bernstein et al. 1987], the scheduler in the DBMS generates only strict schedules—each access (read or write) to a data item by a transaction T is delayed until after the most recent different transaction that wrote to the item has terminated (i.e., either committed or rolled back). Thus if a transaction has not written to the data item, then a read will return the initial value if no committed transaction has written to it; otherwise it will

[4]In ARIES and other WAL protocols, this atomic action is the flushing of the transaction's commit log record.

return the value written by the last transaction that wrote to the item and committed. Finally, if system failures occur during recovery, all transactions that were active (not committed or rolled back) at the time of the failure are rolled back and missing updates from committed transactions are redone. Thus taking system failures into account does not change the *appropriate* value.

In making this natural definition into a precise mathematical form, there are some areas that require care. First, in our description of appropriate values we refer to committed transactions. As stated earlier, our specification is independent of the internal states (the log, stable database, and cache) of the data manager. This causes a complication to our definitions as the external actions alone *do not* contain enough information to determine exactly which transactions have committed. Let $\alpha = S_0\pi_1S_1\pi_2S_2, \ldots, \pi_nS_n$ be any execution. For each transaction T, there are now three cases to consider.

Case 1. If $ReqCommit(T) \notin \alpha$ or $AckRollbk(T) \in \alpha$, then T is not committed in state S_n.

Case 2. If $AckCommit(T) \in \alpha$, then the transaction T is committed.

Case 3. If $ReqCommit(T) \in \alpha$, $AckRollbk(T) \notin \alpha$, and $AckCommit(T) \notin \alpha$, then we *do not* know the status of transaction T as it all depends on which internal actions, if any, have been executed.[5]

Our approach is to allow the protocol some freedom. Our correctness definition is stated using a loosely constrained set of transactions *CommitTrans(α)* that can be regarded as committed such that the set is a subset of all transactions that have requested to commit but their requests did not fail, and a superset of all transactions that have successfully committed. Mathematically, we require that

$$DefinitelyCommitted(\alpha) \subseteq CommitTrans(\alpha) \subseteq PossiblyCommitted(\alpha),$$

where $DefinitelyCommitted(\alpha) = \{T : \exists i,\ \pi_i = AckCommit(T)\}$ and $PossiblyCommitted(\alpha) = \{T : \exists i,\ \pi_i = ReqCommit(T)\} - \{T : \exists j,\ \pi_j = AckRollbk(T)\}$. Also, if the data manager has decided a transaction has committed in state S_i, then it can not change its mind; that is, $\forall j,\ i \leq j \leq n$, the data manager must consider the transaction as committed in state S_j.

The second point requiring care is that in the I/O automaton model, input actions are never blocked, thus they may occur at any time. However, in a real system, normal processing operations such as reads and writes do *not* occur during recovery,[6] or after the requesting transaction has terminated. We introduced a notion of *transaction well-formedness* to capture the

[5]In ARIES and other WAL protocols, if the internal action of flushing the commit log record has occurred then we say T is committed, otherwise it has not committed.

[6]In a real system, users can not access the database during recovery. Hence reads and writes do not occur during this period.

constraints on when requests can be submitted by the scheduler. We then say that the data manager is correct if reads see appropriate values in the well-formed behaviors; arbitrary responses are allowed when requests occur at unreasonable times.

We now give mathematical definitions to these concepts. We first define the three possible alternatives for the status of a transaction. We express this through defining the set of transactions with each status; this set is written as though it were a "history variable" in the system state.

Definition 3.2.1. For any execution $\alpha = S_0\pi_1S_1\pi_2S_2, \ldots, \pi_nS_n$,

(1) $S_k.ActiveTrans$ denotes the set of T for which $\exists a$ such that $a < k$ and $\pi_a \in \{ReqRead(T, x), ReqWrite(T, x, v)\}$ for any data item x and value v, and
$\nexists e$, $a < e < k$, $\pi_e \in \{ReqCommit(T), ReqRollbk(T), Crash\}$.

(2) $S_k.ReqTermTrans$ denotes the set of T such that $\exists t$ such that $t < k$ and $\pi_t \in \{ReqCommit(T), ReqRollbk(T)\}$ or
$\pi_t = Crash$ and $T \in S_{t-1}.ActiveTrans$.

(3) $S_k.TermTrans$ denotes the set of T such that $\exists t$ such that $t < k$ and $\pi_t \in \{AckCommit(T), AckRollbk(T)\}$ or
$\pi_t = AckRestart$ and $T \in S_{t-1}.ReqTermTrans$.

Recovery follows any crash and it rolls back all active transactions. Thus one can view a crash as a rollback request for all active transactions. We have, in the definitions, defined the set of active transactions ($S_k.ActiveTrans$) as those that have requested an access (read or write) but have not requested commit or rollback, nor have they been affected by a crash. The transactions that have requested to terminate ($S_k.ReqTermTrans$) are those which the data manager is rolling back (either due to a rollback request or a crash) or committing. Finally, when the data manager has completed the rollback (either due to a rollback request or a crash) or commit of a transaction, the transaction is then defined as terminated ($S_k.TermTrans$).

We now formalize the requirements on when a transaction may make requests.

Definition 3.2.2. The execution $\alpha = S_0\pi_1S_1\pi_2S_2, \ldots, \pi_nS_n, \ldots,$ is well-formed provided the following hold for every k.

—If $T \in S_k.ReqTermTrans$ then $\forall x, v, k' : (k < k') \Rightarrow$

$$(\pi_{k'} \notin \{ReqRead(T, x), ReqWrite(T, x, v), \quad ReqCommit(T), ReqRollbk(T)\}).$$

—If $\pi_i = ReqRead(T, x)$ and $\pi_j = ReqWrite(T, x, v)$, then

$$(i < j) \Rightarrow (\exists i' : i < i' < j,\ \pi_{i'} = AckRead(T, x))$$

$$(j < i) \Rightarrow (\exists j' : j < j' < i,\ \pi_{j'} = AckWrite(T, x, v)).$$

—If $\pi_k \in \{ReqCommit(T),\ ReqRollbk(T)\}$, then

$$|\{i : i < k \quad \pi_i = ReqRead(T, x)\}| = |\{j : j < k \quad \exists v,\ \pi_j = AckRead(T, x, v)\}|$$

$$|\{i : i < k \quad \pi_i = ReqWrite(T, x, v)\}| = \{j : j < k \quad \pi_j = AckWrite(T, x, v)\}|.$$

—If $\pi_k = ReqWrite(T, x, v)$ then $\forall k', v' : (k < k') \Rightarrow (\pi_{k'} \neq ReqWrite(T, x, v'))$.

—If $\pi_c = Crash$, and $\forall c', ((c < c' < k) \Rightarrow (\pi_{c'} \neq AckRestart))$ then π_k is either *Crash* or else an output or internal action.

The preceding definition formally states the obvious properties of transactions. They are: once a transaction has requested to commit or rollback, or a crash has occurred, it can no longer make any further requests (this also implicitly states that transaction identifiers are unique); reads and writes to the same data item are ordered (one can not issue a read if a previous write to the same data item has not completed and vice versa); all access operations (read and writes) must be completed before a transaction can request to commit or rollback; each transaction can write to a data item at most once;[7] finally, there are no requests from transactions after a crash and before recovery is complete.

Strictness [Bernstein et al. 1987] states that before a transaction can read or write to a data item, the previous transaction that wrote to it must have terminated. The formal definition is shown in Definition 3.

Definition 3.2.3 The execution $\alpha = S_0\pi_1S_1\pi_2S_2 \ldots \pi_nS_n$ is strict provided that for any values v_1, v_2 and data item x, if $k_1 < k_2$, $T_1 \neq T_2$, and

$$\pi_{k_1} = ReqWrite(T_1, x, v_1) \quad \text{and} \quad \pi_{k_2} \in \{ReqWrite(T_2, x, v_2), ReqRead(T_2, x)\},$$

then $\exists k$, $k_1 < k < k_2$ and $\pi_k \in \{AckCommit(T_1), AckRollbk(T_1), AckRestart\}$.

Correctness states that each read to any data item x returns the *appropriate* value. If no transaction has written to x and committed, then we say that the appropriate value is the initial value, and if the transaction currently reading x also wrote to x before the read, then the appropriate value is the value it wrote. In the final case, the appropriate value is defined as the value written by the last transaction that committed and wrote to x. Definition 4 first defines the set of actions, in state S_k of the execution α, that wrote to data item x by a set of transactions $Trans(\alpha)$; then in Definition 5, we define the value that was written by the last action in this set. When we use these definitions in our correctness definition, we take $Trans(\alpha)$ to be the set of committed transactions. Thus we know what is the last committed value with respect to the external actions.

Definition 3.2.4. Let $\alpha = S_0\pi_1S_1\pi_2S_2, \ldots, \pi_nS_n$ be an execution, $1 \leq k \leq n$, and $Trans(\alpha)$ is a set of transactions, then we define

[7]This simplifies many definitions and does not reduce the expressiveness of the model.

$$Updates(\alpha, Trans(\alpha), x, k) = \{i : i \leq k \pi_i$$

$$= AckWrite(T, x, v) \quad \text{and} \quad T \in Trans(\alpha)\}.$$

Definition 3.2.5. Let $\alpha = S_0\pi_1S_1\pi_2S_2, \ldots, \pi_nS_n$ be an execution, then we define

$$LastUpdate(\alpha, Trans(\alpha), x, k) = \begin{cases} v & Updates\ (\alpha, Trans(\alpha), \quad x, k \neq 0, \\ 0 & \text{otherwise} \end{cases}$$

where $1 \leq k \leq n$, $\pi_i = AckWrite(T, x, v)$ and $i = max(Updates(\alpha, Trans(\alpha), x, k))$.

Notice in Definition 3.2.5 that if no transaction in $Trans(\alpha)$ has written to x, then the value is defined as 0.

The final definition now defines correctness of the data manager.

Definition 3.2.6. Given any allowable execution, $\alpha = S_0\pi_1S_1\pi_2S_2\pi_3 \ldots \pi_nS_n$ that is well-formed and strict to the data manager, we say that α is correct provided there exists a set $CommitTrans(\alpha)$ such that

$$\{T : \exists\, i, 1 \leq i \leq n, \quad \pi_i = AckCommit(T)\} \subseteq CommitTrans(\alpha)$$

$$\subseteq \{T : \exists\, i, 1 \leq i \leq n, \quad \pi_i = ReqCommit(T)\} - \{T : \exists j,$$

$$1 \leq j \leq n, \quad \pi_j = AckRollbk(T)\}$$

and $\forall r$, if $\pi_r = AckRead(T, x, v)$, then

$$v = \begin{cases} v' & \text{if } \exists j, j < r, \\ & \pi_j = ReqWrite(T, x, v') \\ LastUpdate(CommitTrans(\alpha), x, r) & \text{otherwise.} \end{cases}$$

We say that the data manager is correct provided that every behavior that is well-formed and strict is correct.

4. MODEL OF THE ALGORITHM

In this section, we present the model of a data manager whose algorithm is based on ARIES [Mohan et al. 1992]. Before we present the model, we first describe the modeling methodology; that is, we explain how we convert the informal description in Mohan et al. [1992] into mathematics. Next we present the model's data structure, which is the formal representation of the state of the data manager at any time. Finally we give the transitions, showing how the state evolves as the data manager runs. This mathematical description should be useful to implementors, even if they are not interested in the verification, because it can clarify subtle alternatives in the algorithm.

While presenting the model, we also give a comprehensive description of the ARIES algorithm as well as the minor simplifications and modifications we make. The model and verification include most of the features provided by ARIES; in particular we model buffer management, write-ahead logging, asynchronous checkpointing, multiple data items per page, and the possibility of crashes during restart processing.[8]

Our model, however, only models physical logging—that is, each log record includes the before and after images of the operation, and the only operations on the data items are reads and writes. Our model does not model the more general type of data managers that provide logical logging which logs the name and arguments of the operation itself. Because we are only concerned with physical logging, we assume that the executions from the scheduler to the data manager are strict [Bernstein et al. 1987] on data items.

The model also makes the assumptions that the database consists of a finite number of data items, each on a fixed page, and that each transaction can update a data item at most once. In addition, once a transaction starts rolling back, it can never commit; that is, our model does not include partial rollbacks.

4.1 Methodology

Our model uses all three types of actions from the I/O automaton model. Input actions are used to model the requests from the scheduler to the data manager. For example, we use the action *ReqRead*(T, x) to model the request from the scheduler to the data manager on behalf of the transaction T to read the value stored in the data item x. Internal actions are used to model the internal steps of the data manager, and the output actions are used to model the acknowledgments, from the data manager to the scheduler, of the completion of a request.

There are three major problems in modeling and verifying a data manager using the I/O automaton model: crashes can occur at any time (even during recovery); there are many concurrent activities occurring in the data manager; and finally, the actions of the I/O automaton model are atomic. Thus each action should contain only one step of the real system. This would lead to a massive model. To make our work tractable, our model allows each action to contain more than one step of the data manager if the following constraints are satisfied.

—There is at most one write to stable storage but any number of reads.

—There is only one write to each shared data structure.

—The sequence of steps is properly latched.

[8]Thus our verification proves that the recovery algorithm is idempotent [Bernstein et al. 1987]—that is, any number of incomplete executions of the recovery algorithm followed by a complete one will restore the database to the same state as a single complete execution.

We claim that our model does not miss anything significant for the following reasons.

—A crash has the property that it corrupts the contents of volatile main memory but has no effect on stable disk storage. Therefore if a crash interrupts a sequence of steps (from one action in the formal model) that only access volatile storage or read from stable storage, then the effect is the same as if all the steps never occurred, because all the alterations to the state are lost anyway. We assume that writes to stable storage are atomic. If a sequence of steps containing only one write to stable storage is interrupted by a crash, then its effect is equivalent to either the steps not occurring at all (when the crash occurred before the write) or the steps all completed followed by a crash (when the crash occurred after the write). This is why we allow only one write to stable storage but any number of reads in one action of the model.

—If an action contained more than one write to the same data structure (e.g., data item or an entry in the transaction table) in volatile storage, then our model would be incomplete as only the value written by the last writer in the sequence can be flushed to stable storage, whereas in reality stable storage might contain an intermediate value. Therefore we only allow one write to each data structure within an action of the model.

—There are concurrent activities within the data manager. One transaction may be rolling back while another is making forward progress (doing reads and updates), for example. These steps may interfere with each other when they access shared memory (e.g., transaction table, log, and data items) and cause physical inconsistencies [Gray and Reuter 1993]. In real systems, latches are used to prevent these inconsistencies by making a sequence of steps look atomic. Using the I/O automata model, we do not need to model latches as the actions are already atomic but need to model all the allowable concurrency. Therefore we require that each action contain only steps of the real system that are properly latched.

4.2 Data Structures

This section describes and lists the data structures used in our model. The main data structures in ARIES are the log, transaction table, dirty pages table, and the pages that contain the data items.

4.2.1 *Log.* The log is modeled as an array of log records indexed by log sequence numbers (LSN). In the following we describe the fields in the log.

LSN: The log sequence number of the log record.

Type: Indicates the type of the log record. There are six types: *update*, *compensate*, *end*, *begin checkpoint*, *end checkpoint*, and *operating system file return* (*OsFileReturn*).

TransID: Records the identifier of the transaction if the log record is a transaction related log record (log record of type *update*, *compensate*, or *end*).

PrevLSN: This field is only used for transaction related log records and points to the previous log record written by the same transaction. Thus the log records written by a transaction are backwardly linked via this pointer. If it is the first log record from a transaction, then it points to the start of the log (*PrevLSN* = 0).

PageID: This field is only used for log records of type *update*, *compensate*, and *OsFileReturn*. It records the page identifier of the page that the recorded operation affected.

UndoNxtLSN: This field is only used in *compensation* log records and it points to the next log record to be rolled back for the transaction; that is, it points to the previous update, by the same transaction, to the update that was undone by the compensation action that is recorded in this log record. If all updates by the transaction have been undone, then this points to the beginning of the log.

Data: This is a variant record. For *update* and *compensation* log records, it records the undo/redo information that is the before and after images of the operation.[9] For *end* log records, it records the termination of a transaction. In this field, we record the final status of the transaction—committed or rolled back. For *end checkpoint* log records, it records checkpoint related information (checkpoint's copy of the transaction table and dirty pages table). For all other types of log records, this field is left empty.

In the real system, there are two logs. Older log records are stored in a log on disk, and the most recent records are kept in a buffer (volatile storage); from time to time, entries are moved from the buffer and appended to the log on disk. In our model we have a single array that contains all log records, and represents the concatenation of these two real structures. We use the variable *LSN* to point to the next free log location for the volatile front of the log, and we use *StableLSN* to point to the first entry that is in the volatile buffer. Thus all log records whose log sequence number is strictly less than *StableLSN* are in stable storage. After a crash, the analysis pass starts scanning the log from the log record pointed to by the *master record* which is stored in some well known location in stable store. We use the variable *MasterRec* to model the master record. All the data structures related to the log are shown in Figure 1.

4.2.2 *Transaction Table.* The transaction table keeps track of the status of transactions and is represented as an array of transaction table entries indexed by transaction identifiers. Each active transaction has an entry in the transaction table and each entry contains two pointers: one points to the last record written by the transaction (*LastLSN*) and the other is to the next log record to be undone in case of a rollback (*UndoNxtLSN*). The data structures are shown in Figure 2.

[9]As a *compensation* log record is never undone, in fact only the after image is needed. We include both images for uniformity.

```
LSN:        Log sequence number;
StableLSN:  Log sequence number;
MasterRec:  Log sequence number;
LogRec:     Record
    LSN:  Log sequence number;
    Type: {Update, Comp, End, BgnCheckpt, EndCheckpt, OsFileReturn};
    TransID:        Transaction identifier;
    PrevLSN:        Log sequence number;
    UndoNxtLSN:     Log sequence number;
    PageID:         Page identifier;
    Data: Case (Type) of
            Update or Comp: Record
                DataItemName:  Name of the data item;
                BeforeImage:   value;
                AfterImage:    value;
            endrecord;
            End:
                Status:        {Commit, Rollback};
            EndCheckpt: Record
                TransTable:    Transaction Table; /* see below */
                DirtyPages:    Dirty Pages Table; /* see below */
            endrecord;
    endcase;
endrecord;
Log:    array of LogRec indexed by LSN;
```

Fig. 1. Log.

```
TransTableEntry: Record
    TransID:    Transaction identifier;
    LastLSN:    Log sequence number;
    UndoNxtLSN: Log sequence number;
endrecord;
TransTable:    array of TransTableEntry indexed by TransID;
```

Fig. 2. Transaction table.

4.2.3 *Dirty Pages Table.* The dirty pages table is used to keep track of the dirty pages in the DBMS. The table is represented as an array of dirty page entries indexed by page identifiers. Each entry contains a single pointer that points to the log record that describes the first operation that changed the page since it was last brought into cache. The data structures are shown in Figure 3.

4.2.4 *Page.* Each page contains a set of data items and each data item has an associated value (the value stored in the data item). Also, each page records in the field *PageLSN* the log sequence number of the last log record corresponding to an action whose effects are reflected in the page. A copy of each page is maintained in stable storage and a subset of them are also in volatile cache. Both the stable and cache pages are represented by arrays of pages indexed by page identifiers. We represent a page which is not in cache by assigning $\emptyset$ to the page's cache entry. The data structures are shown in Figure 4.

Thus the value stored, in state S_n, in data item x is represented as $S_n.Stable[P].x$ where P is the page identifier of the page where x resides

```
DirtyPagesEntry: Record
    PageID:    Page identifier;
    RecLSN:    Log sequence number;
endrecord;
DirtyPages:    array of DirtyPagesEntry indexed by PageID;
```

Fig. 3. Dirty pages table.

```
Page: Record
    PageID:     Page identifier;
    PageLSN:    Log sequence number;
    x1:       value;
    x2:       value;
    .
    .
    .
    xn:       value;
endrecord;

Stable:   array of pages indexed by PageID;
Cache:    array of pages indexed by PageID;
```

Fig. 4. Page.

and similarly for the cache. Also, we represent a page P as not in cache by $S_n.Cache[P] = \emptyset$. In the model, we use the notation *Page*(x) to be the page identifier of the page where x resides.

4.2.5 *Checkpoints.* Checkpoints are taken asynchronously and they take a copy of both the transaction table and the dirty pages table. The data structures used by checkpointing are, therefore, a transaction table and a dirty pages table. These data structures are shown in Figure 5.

4.2.6 *Recovery.* During recovery, we use a number of variables to keep track of the process. We use the variables *AnalysisLSN, RedoLSN*, and *UndoLSN* to respectively point to the next log record the analysis pass, redo pass, and undo pass have to process. Recovery needs to keep track of which transactions have terminated and which pages have been flushed. These are recorded in *TerminatedTrans* and *FlushedPages*. Finally, we use a Boolean variable to record if we have already processed the last successful checkpoint's *end checkpoint* log record. The data structures are shown in Figure 6.

4.2.7 *System State and Control Flow.* The final set of data structures in our model is used to show the current state of the DBMS and for the control flow of the model. The variable *SystemState* indicates whether the system is *down*, *up*, or *recovering*, and the set *ActiveSet* contains a set of internal action names. In general, an internal action is enabled if it is an element in the set.[10] An action may enable another action by inserting the action name into the set and an action can disable itself by removing its own name from the set. Thus an action may be enabled in state S_i but disabled in the following state. The data structures are shown in Figure 7.

[10]Input actions are always enabled so are never in this set.

```
CheckptTransTable:   array of TransTableEntry indexed by TransID;
CheckptDirtyPages:   array of DirtyPagesEntry indexed by PageID;
```

Fig. 5. Checkpoints.

```
AnalysisLSN: Log sequence number;   TerminatedTrans: Set of transaction identifiers;
RedoLSN:     Log sequence number;   FlushedPages:    Set of page identifiers;
UndoLSN:     Log sequence number;   FirstCheckpt:    Boolean;
```

Fig. 6. Recovery.

```
SystemState:  {Down, Normal, Recovery};
ActiveSet:    Set of action names;
```

Fig. 7. System and control flow.

4.3 The Actions

In this section we describe the input, internal, and output actions (preconditions and effects) that model the way the data manager changes state as it executes. A crash terminates all activities and from Definition 3.2.2 on well-formed transactions, it follows that the actions for *Read*, *Write*, *Commit*, and *Rollback* are never enabled during recovery. As a result, there is no need to explicitly include the condition that the system must be in normal processing for these actions to be enabled.

4.3.1 *Read.* The actions associated with read are modeled by an input action *ReqRead*(T, x), an internal action *Read*(T, x), and an output action *AckRead*(T, x, v). The input action enables the internal action that reads the value and enables the output action. The output action is the acknowledgment of the completion of the read.

One of the preconditions to the internal action is that the appropriate page is in the cache. Our model does not take specific action to bring it in, but the *Fetch*(P) action can occur at any time and it will cause a copy of the page from stable storage to be in the cache. That is, our model uses nondeterminism to be very general. This could affect the liveness, but we do not mind, inasmuch as our proof only deals with safety (i.e., it shows that any value returned is correct). The actions are shown in Figure 8.

4.3.2 *Write.* Again, three actions are used to model a write. The internal action *Write*(T, x, v) models all of the following: altering the cache, adding a record to volatile log, and updating the transaction table and dirty pages table. In a real system, a latch on the page where x resides is held during these steps, another latch on the transaction table is held during the log write and the subsequent modification of T's entry in the transaction table, and a latch on the log is held during the log write. Thus this sequence of steps is properly latched and there are no accesses to stable storage. Also, each data structure (each entry in the transaction table, dirty pages table, and volatile log) is only written to once. Hence we can model these steps in a single action. The actions are shown in Figure 9. In the action *Write*(T, x, v), if the page is not dirty, then an entry in the dirty

```
ReqRead(T,x) [Input]
Effects:
  ActiveSet = ActiveSet ∪ {Read(T,x)};
Read(T,x) [Internal]
Precondition:
  Read(T,x) ∈ ActiveSet; /* page is in cache */
  Cache[Page(x)] ≠ ∅;
Effects:
  v = Cache[Page(x)].x; /* Reads the value */
  ActiveSet = (ActiveSet - Read(T,x)) ∪ {AckRead(T,x,v)};
AckRead(T,x,v) [Output]
Precondition:
  AckRead(T,x,v) ∈ ActiveSet;
Effects:
  ActiveSet = ActiveSet - {AckRead(T,x,v)};
```

Fig. 8. Read actions.

pages table is inserted as the write has now caused the page to become dirty. An *update* log record is written to record the *write* to the database and the transaction table is updated to record that this is now the last log record written by *T* and the first to be undone in case of a rollback. We leave the *UndoNxtLSN* field of the log empty as only *compensation* log records use this field. Finally, the value is written to the cache's copy of the data item and the page's *PageLSN* is updated so that it records that this write is now the last action that affected the page.

4.3.3 *Commit.* When the data manager receives a commit request, it tries to commit the transaction by ensuring that the updates, made by the transaction, are durable. In ARIES, this is achieved by enforcing the redo rule [Bernstein et al. 1987]; that is, write a commit log record for the transaction and flush all log records written by the transaction to the stable log. Log flushes are sequential; therefore when the transaction's *commit* log record is in stable storage we know that the redo rule is satisfied. Hence the atomic action that commits a transaction is the log flush of the transaction's commit log record.

We model the commit procedure using the actions that are shown in Figure 10. The action *Commit*(*T*) models the data manager writing an *end commit* log record and the action *FailCommit*(*T*) models the data manager rejecting a commit request. Note that the acknowledgment to a commit is only enabled after the transaction's *commit* log record is in the stable log. Thus flushing is not explicitly done by these actions, but rather it is done nondeterministically by the action that flushes the log record (see later). Also, notice that *ComLSN* (in the code for *AckCommit*(*T*) and other commit related actions) is a free variable not a state component; the meaning of such code is that there should exist some value for *ComLSN* to satisfy the preconditions, and this value is not used in the effects activity.

4.3.4 *Rollback.* When a transaction *T* rolls back, the system needs to undo all its updates in reverse chronological order. ARIES writes a *compensation* log record to record each undo. All log records written by a transac-

```
ReqWrite(T,x,v) [Input]
Effects:
  ActiveSet = ActiveSet ∪ {Write(T,x,v)};
Write(T,x,v) [Internal]
Precondition:
  Write(T,x,v) ∈ ActiveSet;
  Cache[Page(x)] ≠ ∅;
Effects:
  ActiveSet = (ActiveSet - Write(T,x,v)}) ∪ {AckWrite(T,x,v)};
  if ( DirtyPages[Page(x)] = ∅ )
    DirtyPages[Page(x)].RecLSN = LSN;
  endif;
  Log[LSN].Type = Update;
  Log[LSN].TransID = T;
  Log[LSN].PageID = Page(x);
  Log[LSN].Data.DataItemName = x;
  Log[LSN].Data.BeforeImage = Cache[Page(x)].x;
  Log[LSN].Data.AfterImage = v;
  TransTable[T].LastLSN = LSN;
  TransTable[T].UndoNxtLSN = LSN;
  Cache[Page(x)].x = v; /* Updates the database */
  Cache[Page(x)].PageLSN = LSN;
  LSN = LSN + 1;
AckWrite(T,x,v) [Output]
Precondition:
  AckWrite(T,x,v) ∈ ActiveSet;
Effects:
  ActiveSet = ActiveSet - {AckWrite(T,x,v)};
```

Fig. 9. Write actions.

tion are backwardly linked by the *PrevLSN* pointer in each log record. The rollback procedure repeatedly undoes the update described in the log record pointed to by *TransTable*[*T*].*UndoNxtLSN*, until *TransTable*[*T*].*UndoNxtLSN* = 0.

Initially (before the start of the rollback procedure)

$$TransTable[T].UndoNxtLSN = TransTable[T].LastLSN,$$

and each time an update is undone, the *UndoNxtLSN* is set to point to the log record that recorded the previous update to the update that has just been undone by *T*. Thus when the rollback procedure terminates, all updates by the transaction will have been undone. An *update* is undone by the data manager by restoring the before image of the *update*. A *compensation* log record is written to record the undo and the transaction table is updated to reflect the new state of the transaction; that is, the next *update* to be undone is the *update* recorded in the log record pointed to by the *PrevLSN* field of the *update* log record just undone and *T*'s *LastLSN* entry in the transaction table now points to the *compensation* log record that was just written.

The procedure to roll back a transaction is complex to model due to the loop that undoes all the updates in reverse chronological order. We use the action *Rollback*(*T*) to simulate the undo of a single log record's update; the

```
ReqCommit(T) [Input]
Effects:
  ActiveSet = ActiveSet ∪ {Commit(T)};
Commit(T) [Internal]
Precondition:
  Commit(T) ∈ ActiveSet;
Effects:
  ActiveSet = (ActiveSet - {Commit(T)})
    ∪ {AckCommit(T, LSN)}
  Log[LSN].Type = End;
  Log[LSN].TransID = T;
  if (TransTable[T] = ∅)
    Log[LSN].PrevLSN = 0;
  else
    Log[LSN].PrevLSN =
      TransTable[T].LastLSN;
  endif;
  Log[LSN].Data = Commit;
  TransTable[T] = ∅;
  LSN = LSN + 1;
```

```
FailCommit(T) [Internal]
Precondition:
  Commit(T) ∈ ActiveSet;
Effects:
  ActiveSet = (ActiveSet ∪
     {Rollbk(T)}) - {Commit(T)};
AckCommit(T) [Output]
Precondition:
  AckCommit(T, ComLSN) ∈ ActiveSet;
  /* Satisfies the redo rule */
  StableLSN > ComLSN;
Effects:
  ActiveSet = ActiveSet -
     {AckCommit(T, ComLSN)};
```

Fig. 10. Commit actions.

action *RollbackTerm*(*T*) then terminates the rollback procedure when all the updates by the transaction *T* have been undone. The action *Rollback*(*T*) occurs a number of times in the execution when a transaction rolls back. To be more precise, the number is exactly the number of *update* log records the transaction has written to the log.

Notice that, unlike the commit procedure, the rollback can acknowledge the completion of the rollback as soon as the *end rollback* log record is written to volatile log. The actions are shown in Figure 11. In the action *Rollbk*(*T*), we use the symbols UndoingLSN, P and x as a shorthand for the values expressed in the precondition.

4.3.5 *Checkpoints.* In ARIES, checkpoints are taken periodically. They can be taken asynchronously during normal processing: the redo pass and the undo pass. However, they can not be taken during the analysis pass because both the transaction table and dirty pages table are not up to date (correct). Checkpoints take a copy of the transaction table and dirty pages table, and record them in a log record. Recovery, which needs to reconstruct these tables, uses this information to reduce the number of log records it needs to process thus reducing recovery time. A checkpoint first writes a *begin checkpoint* log record to the log, then takes a copy of the dirty pages table and transaction table. It then writes an *end checkpoint* log record, which contains a copy of the checkpoint's copy of the dirty pages table and transaction table, to the log. Finally, after the *end checkpoint* log record has been flushed, the *master record* is updated such that the *LSN* in the *master record* points to the checkpoint's *begin* log record.

The paper [Mohan et al. 1992] describes how the checkpoint takes a copy of the dirty pages table—that is, the system repeatedly latches a number of rows[11] of the table, takes a copy of them, and then unlatches them until a copy of every entry in the table has been taken. Note that after the checkpoint unlatches the rows, the entries in the dirty pages table may change or even be deleted (perhaps the page was flushed so it was no longer dirty). Thus the checkpoint's copy of the dirty pages table may be out of date when the *end checkpoint* log record is written. The recovery algorithm takes this into account when it reconstructs the table. However, Mohan et al. [1992] do not describe how the checkpoint takes a copy of the transaction table. We have naturally assumed that the procedure for the transaction table is the same as the one for the dirty pages table. As a result, we had to modify the algorithm for the analysis pass of recovery. We discuss in detail in Section 4.3.8 these modification and why they were required.

The actions that model the checkpoint procedure are shown in Figure 12. Notice we only allow one checkpoint to be active at any time. Also, the action *EndCheckpt*(*BLSN*), which models the checkpoint procedure writing the *end checkpoint* log record, is enabled if

$$\{T\colon \mathit{CheckptTransTable}[T] \neq \emptyset\} \supseteq \{T : \mathit{TransTable}[T] \neq \emptyset\} \quad \text{and}$$

$$\{P\colon \mathit{CheckptDirtyPages}[P] \neq \emptyset\} \supseteq \{P : \mathit{DirtyPages}[P] \neq \emptyset\}.$$

That is, at the end of the checkpoint, each nonempty entry in the transaction table has an entry in the checkpoint's copy of the table. Because checkpoints are taken asynchronously, it is possible for a transaction to terminate after the checkpoint has taken a copy of the transaction's entry in the transaction table. Thus the transaction will not have an entry in the transaction table while having an entry in the checkpoint's copy of the transaction table; similarly for the dirty pages table.

There are five actions that represent various activities with checkpointing: the action *BgnCheckpt* writes a begin checkpoint log record; the actions *CheckptTransTable*(*T*), *CheckptDirtyPages*(*P*) model the checkpoint taking a copy of a row of the transaction table and dirty pages table, respectively. When copies of all rows of both tables have been taken, an *end checkpoint* log record, with the tables, is written to the log by the action *EndCheckpt*-(*BLSN*). The action *WriteMaster*(*BLSN*) then writes the *LSN* of the checkpoint's begin log record to the master record. A precondition is that the *end checkpoint* log record, written by the action *EndCheckpt*(*BLSN*), is in the stable log.

The actions are shown in Figure 12 and we define the set *CheckptActions* to be the set of actions that model asynchronous checkpointing.

4.3.6 *Flush and Fetch.* Pages are continuously fetched and flushed by the DBMS and log records are also continuously flushed to the stable log. The actions that model the fetch and flush actions are shown in Figure 13. Note that when a page is flushed during normal processing and the undo

[11]A row contains a single page's entry in the table.

```
ReqRollbk(T) [Input]
Effects:
  ActiveSet = ActiveSet ∪ {Rollbk(T)};
Rollbk(T) [Internal]
Precondition:
  let P = TransTable[T].PageID;
  Rollbk(T) ∈ ActiveSet;
  TransTable[T] ≠ ∅;
  TransTable[T].UndoNxtLSN > 0;
  Cache[P] ≠ ∅;
Effects:
  let UndoingLSN =
    TransTable[T].UndoNxtLSN;
  let x =
    Log[UndoingLSN].Data.DataItemName;
  if (DirtyPages[P] = ∅)
    DirtyPages[P] = LSN;
  endif;
  Log[LSN].Type = Comp;
  Log[LSN].TransID = T;
  if TransTable[T] ≠ ∅
    Log[LSN].PrevLSN = 0;
  else
    Log[LSN].PrevLSN =
      TransTable[T].LastLSN;
  endif;
  Log[LSN].UndoNxtLSN =
    Log[UndoingLSN].PrevLSN
  Log[LSN].PageID = P;
  Log[LSN].Data.DataItemName = x;
  Log[LSN].Data.AfterImage =
    Log[UndoingLSN].Data.BeforeImage
  TransTable[T].LastLSN = LSN;
  TransTable[T].UndoNxtLSN =
    Log[UndoingLSN].PrevLSN;
  Cache[P].x =
    Log[UndoingLSN].Data.BeforeImage;
  Cache[P].LSN = LSN;
  LSN = LSN + 1;
RollbkTerm(T) [Internal]
Precondition:
  Rollbk(T) ∈ ActiveSet;
  (TransTable[T].UndoNxtLSN = 0 or
    TransTable[T] = ∅);
Effects:
  ActiveSet = (ActiveSet - {Rollbk(T)})
      ∪ {AckRollbk(T)};
  Log[LSN].Type = End;
  Log[LSN].TransID = T;
  if (TransTable[T] ≠ ∅)
      Log[LSN].PrevLSN =
          TransTable[T].LastLSN;
  else
    Log[LSN].PrevLSN = 0;
  endif;
  Log[LSN].UndoNxtLSN = 0;
  Log[LSN].Data = Rollback;
  TransTable[T] = ∅;
  LSN = LSN + 1;
AckRollbk(T) [Output]
Precondition:
  AckRollbk(T) ∈ ActiveSet;
Effects:
  ActiveSet = ActiveSet - {AckRollbk(T)};
```

Fig. 11. Abort actions.

pass of recovery, it is recorded in the log and the page's entry in the dirty pages table is removed as the page is no longer dirty. However, the entry is not removed when the flush occurs during the redo pass because the effects of some operations that are recorded in the later part of the log may still require redo (they are still missing from the DBMS). If the page's entry was removed, then the redo pass would not redo these updates. Therefore recovery would be incorrect. When a flush occurs during the redo pass, all we know is that all operations whose corresponding log record's *LSN* is less than *RedoLSN* are not missing from the DBMS and we update the dirty pages to reflect this.

The action *Flush*(*P*) models the flush of a page from volatile storage to stable storage by copying the contents of *Cache*[*P*] to *Stable*[*P*]. One of its preconditions is *Cache*[*P*].*PageLSN* < *StableLSN*. This condition enforces

```
BgnCheckpt [Internal]
Precondition:
  (SystemState = Normal or
    SystemState = Recovery);
  Analysis ∉ ActiveSet;
  CheckptActions ∩ ActiveSet = ∅;
Effects:
  ActiveSet = ActiveSet ∪
    {CheckptTransTable,
      CheckptDirtyPages,
      EndCheckpt(LSN)};
  CheckptTransTable = ∅;
  CheckptDirtyPages = ∅;
  Log[LSN].Type = BgnCheckpt;
  LSN = LSN + 1;
CheckptTransTable(T) [Internal]
Precondition:
  CheckptTransTable ∈ ActiveSet;
  CheckptTransTable[T] = ∅;
  TransTable[T] ≠ ∅;
Effects:
  CheckptTransTable[T] = TransTable[T];
CheckptDirtyPages(P) [Internal]
Precondition:
  CheckptDirtyPages ∈ ActiveSet;
  CheckptDirtyPages[P] = ∅;
  DirtyPages[P] ≠ ∅;
Effects:
  CheckptDirtyPages[P] = DirtyPages[P];
EndCheckpt(BLSN) [Internal]
Precondition:
  EndCheckpt(BLSN) ∈ ActiveSet;
  {T:CheckptTransTable[T] ≠ ∅} ⊇
    {T : TransTable[T] ≠ ∅};
  {P:CheckptDirtyPages[P] ≠ ∅} ⊇
    {P: DirtyPages[P] ≠ ∅};
Effects:
  Log[LSN].Type = EndCheckpt;
  Log[LSN].Data.TransTable =
    CheckptTransTable;
  Log[LSN].Data.DirtyPages =
    CheckptDirtyPages;
  LSN = LSN + 1;
  ActiveSet = (ActiveSet ∪
    {WriteMaster(BLSN, LSN)}) -
      {CheckptTransTable,
        CheckptDirtyPages,
        EndCheckpt(BLSN)};
WriteMaster(BLSN, ELSN) [Internal]
Precondition:
  WriteMaster(BLSN, ELSN) ∈
    ActiveSet;
  StableLSN > ELSN
Effects:
  MasterRec = BLSN;
```

Fig. 12. Checkpoint actions.

write-ahead logging. The action *Fetch*(*P*) models the cache manager fetching the page. It copies the contents of *Stable*[*P*] to *Cache*[*P*]. Notice that in our model, the *LogFlush*(*StableLSN*) action, which represents flushing the log record, simply involves incrementing *StableLSN* rather than actually copying log records. Also, page fetches and flushes are not enabled during the analysis pass because the pass does not access any pages.

4.3.7 *Crash.* When a crash occurs in a real system, the contents of volatile storage are lost. That is, the transaction table, the dirty pages table, all the pages in cache, and the volatile log are lost. To model this, all pages in cache and all entries in both tables are set to $\emptyset$, all log records whose log sequence number is greater than or equal to *StableLSN* are set to $\emptyset$,[12] and *LSN* = *StableLSN*. Finally, the *ActiveSet* contains the one element *Restart* which enables the restart procedure. The action is shown in Figure 14.

[12]We set these log entries to $\emptyset$ to simplify some of the arguments in our verification. This is a reasonable reflection of the fact that in a real system, these records are lost after a crash.

```
Flush(P) [Internal]
Precondition:
  SystemState ≠ Down;
  Analysis ∉ ActiveSet;
  /* enforces WAL */
  Cache[P].PageLSN < StableLSN;
Effects:
  Stable[P] = Cache[P];
  Cache[P] = ∅;
  if Redo ∉ ActiveSet
    Log[LSN].Type = OsFileReturn;
    Log[LSN].PageID = P;
    LSN = LSN + 1;
    DirtyPages[P] = ∅;
  else if ((DirtyPages[P] ≠ ∅)
      & (DirtyPages[P].RecLSN < RedoLSN))
    DirtyPages[P].RecLSN = RedoLSN - 1;
  endif;
```

```
Fetch(P) [Internal]
Precondition:
  SystemState ≠ Down;
  Cache[P] = ∅;
Effects:
  Cache[P] = Stable[P];
LogFlush(StableLSN) [Internal]
Precondition:
  SystemState ≠  Down;
  StableLSN < LSN;
Effects:
  StableLSN = StableLSN + 1;
```

Fig. 13. Fetch and flush actions.

4.3.8 *Restart.* The restart procedure recovers the DBMS from a crash. ARIES achieves recovery using three passes of the log: the analysis pass, the redo pass, and the undo pass.

Analysis pass. The analysis pass reconstructs the transaction table and the dirty pages table that were lost in the crash. The dirty pages table (respectively, the transaction table) will then enable the redo (respectively, the undo) pass to determine which logged operations require redo (respectively, undo). The analysis pass scans the log forward from the log record pointed to by the *LSN* stored in the *master record* to the end of the log. When the pass processes an *update*, *compensation*, *end*, or *OsFileReturn* log record, the effects on the transaction table and the dirty pages table are similar to when the log record was initially written. In addition, when the pass processes an *end* transaction log record, it records the termination of the transaction in the set *TerminatedTrans*; similarly for flushed pages. At the end of the analysis pass, an *end rollback* log record is written for each transaction that has an entry in the transaction table and its *UndoNxtLSN* points to the beginning of the log ($TransTable[T].UndoNxtLSN = 0$). Each of these transactions has already been completely rolled back but its *end rollback* log record was lost due to the crash, so we just rewrite these records. The action *Restart* initializes some recovery variables and enables the analysis pass. The internal action *Analysis* is repeated once for each record being processed. Finally the *AnalysisTerm* action enables the redo pass and initializes *RedoLSN* so the redo pass knows where to start processing the log.

We have altered the algorithm for the analysis pass from that described in Mohan et al. [1992] so that when it processes an *end checkpoint* log record, it reconstructs the transaction table using the pseudocode shown in Figure 15. In this, let *LogRec* be the *end checkpoint* log record the analysis pass is currently processing and let the set *TerminatedTrans* contain the

```
Crash [Input]
Effects:
  /* Disables everything else */
  ActiveSet = {Restart};
  SystemState = Down;
  for each transaction T
    TransTable[T] = ∅;
  endfor;
  for each page P
    DirtyPages[P] = ∅;
    Cache[P] = ∅;
  endfor;
  for l = StableLSN to LSN - 1
    Log[l] = ∅;
  endfor;
  LSN = StableLSN;
```

Fig. 14. Crash action.

set of transactions for which the analysis pass has processed the transaction's *end* log record (i.e., these are the transactions that terminated during the checkpoint).

In contrast, in Mohan et al. [1992] the process only checks that the transaction does not have an entry in the transaction table before inserting one. Under our assumption that an entry in the transaction table is unlatched after the checkpoint has taken a copy of it (thus the transaction could commit after the checkpoint has taken a copy of the transaction's entry in the transaction table and before the checkpoint terminates), it is crucial to check that the transaction did not terminate during the checkpoint. The pseudocode in Figure 15 guarantees that terminated transactions do not have an entry in the reconstructed transaction table. In Appendix A we illustrate why the modification is required. Also, in our model, we made a similar alteration for the reconstruction of the dirty pages table. The actions for the analysis pass are shown in Figures 16 and 17, respectively.

Redo pass. The goal of the redo pass is to redo the effects of all missing operations that were lost in the crash (even operations from transactions that did not commit). It scans the log forward, starting from the minimum recovery *LSN* in the dirty pages table to the end of the log. We prove in our verification later that the effects of any action, whose corresponding log record's *LSN* is less than the minimum recovery *LSN* in the dirty pages table, is not missing from the DBMS. When the redo pass processes a redoable (an *update* or *compensation*) log record, the pass examines the dirty pages table to determine if the effects of the recorded operation are potentially missing. If the page that the operation affected is dirty and if the page's recovery *LSN* entry in the dirty pages table is less than or equal to the log record's *LSN*, then the effects of the recorded operation are potentially missing (this is again proved in our verification). For each of these recorded operations, the page the operation affected is fetched, if it is not already in cache. Now, if the *PageLSN* is less than the log record's *LSN*, then the operation requires redo and is redone by copying the after image of the operation to the data item; otherwise it does not require redo and we update the dirty pages table such that the table is brought up to date.

The actions for the redo pass are shown in Figure 18. The *Redo* action is repeated once for each log record processed in this pass; the *RedoTerm* action terminates the redo pass and enables the undo pass.

```
For each transaction T in LogRec.TransTable
    if ((TransTable[T] = ∅) and (T ∉ TerminatedTrans))
        TransTable[T] = LogRec.Data.TransTable[T];
    endif;
endfor;
```

Fig. 15. Pseudocode.

```
Restart [Internal]
Precondition:
  Restart ∈ ActiveSet;
Effects:
  ActiveSet = {Analysis};
  SystemState = Recovery;
  AnalysisLSN = MasterRec;
  TerminatedTrans = ∅;
  FlushedPages = ∅;
  FirstCheckpt = True;
```

Fig. 16. Restart actions.

Undo pass. The undo pass undoes all the updates from loser transactions (these are the transactions that did not commit before the crash). While the transaction table is not empty, the undo pass undoes the update recorded in the log record pointed to by the maximum *UndoNxtLSN* entry in the transaction table. The procedure to undo an update is identical to the undo during a transaction rollback. When all updates by a transaction have been undone, an *end rollback* log record is written to the log and the transaction's entry in the transaction table is removed. Thus, eventually, the transaction table will become empty and the undo pass will terminate.

The actions that model the undo pass are shown in Figure 19. The action *Undo*1 and *Undo*2(U) each represent part of the processing of a log record. These could have been combined to form one action, but we chose to use two as it simplifies our verification in Section 5.

4.3.9 *Start State.* The start state reflects the state of the database at the time it was first installed; that is, all cache slots, the transaction table, and dirty pages table are all empty and the value in the database for each location is 0. To reduce the complexity of the lemmas and proofs, we define the start state of the stable log to contain an *update* log record by the transaction t_0 which writes a value of 0 to a phantom location px.[13] The transaction table and dirty pages table are both empty. The variable *SystemState* is *Normal* and the master record *MasterRec* = 1. Also, *ActiveSet* = $\emptyset$. The start state of the model is shown in Figure 20.

5. VERIFICATION

In this section we verify the recovery algorithm based on ARIES, by showing that the model from Section 4 meets the correctness condition

[13]This log record is used only in definitions; it is never examined by any action.

```
Analysis [Internal]
Precondition:
  Analysis ∈ ActiveSet;
  AnalysisLSN < LSN;
Effects:
  case (Log[AnalysisLSN].Type)
    Update | Comp:
      let T = Log[AnalysisLSN].TransID and P = Log[AnalysisLSN].PageID;
      TransTable[T].LastLSN = AnalysisLSN;
      if (Log[AnalysisLSN].Type = Update )
        TransTable[T].UndoNxtLSN = AnalysisLSN;
      else TransTable[T].UndoNxtLSN = Log[AnalysisLSN].UndoNxtLSN;
      endif;
      if (DirtyPages[P] = ∅ )
        DirtyPages[P].RecLSN = AnalysisLSN;
      endif;
    end;
    EndCheckpt:
      if (FirstCheckpt = True)
        for each T' such that Log[AnalysisLSN].Data.TransTable[T'] ≠ ∅
          if ((TransTable[T'] = ∅) and (T' ∉ TerminatedTrans))
            TransTable[T'] = Log[AnalysisLSN].Data.TransTable[T'];
          endif;
        endfor
        for each P' such that Log[AnalysisLSN].Data.DirtyPages[P'] ≠ ∅
          if (P' ∉ FlushedPages) and ((DirtyPages[P'] = ∅) or
            (DirtyPages[P'].RecLSN > Log[AnalysisLSN].Data.DirtyPages[P'].RecLSN))
              DirtyPages[P'] = Log[AnalysisLSN].Data.DirtyPages[P'];
          endif;
        endfor;
        FirstCheckpt = False;
    end;
    End:
      TransTable[Log[AnalysisLSN].TransID] = ∅;
      TerminatedTrans = TerminatedTrans ∪ {Log[AnalysisLSN].TransID};
    end;
    OsFileReturn:
      FlushedPages = FlushedPages ∪ {Log[AnalysisLSN].PageID};
      DirtyPages[Log[AnalysisLSN].PageID] = ∅;
    end;
  endcase;
  AnalysisLSN = AnalysisLSN + 1;
AnalysisTerm [Internal]
Precondition:
  Analysis ∈ ActiveSet;
  AnalysisLSN = LSN;
Effects:
  ActiveSet = {Redo};
  /* all the updates from these transactions have already been undone */
  for each T such that TransTable[T].UndoNxtLSN = 0
    TransTable[T] = ∅;
    Log[LSN].Type = End; Log[LSN].TransID = T;
    Log[LSN].PrevLSN = TransTable[T].LastLSN;
    Log[LSN].Data = Rollback;
    LSN = LSN + 1;
  endfor;
  RedoLSN = min({DirtyPages[P'].RecLSN: DirtyPages[P'] ≠ ∅} ∪ {LSN});
```

Fig. 17. Analysis pass actions.

```
Redo [Internal]
Precondition:
  Redo ∈ ActiveSet;
  RedoLSN < LSN;
  /* The page is in cache if the page needs to be accessed. */
  (Log[RedoLSN].Type = Update | Comp) ⇒ (Cache[Log[RedoLSN].PageID] ≠ ∅)
Effects:
  if (Log[RedoLSN].Type = Update | Comp)
    let P = Log[RedoLSN].PageID;
    if ((DirtyPages[P]≠ ∅) and (RedoLSN ≥ DirtyPages[P].RecLSN))
      if ( Cache[P].PageLSN < RedoLSN )
        let x = Log[RedoLSN].Data.DataItemName;
        Cache[P].x = Log[RedoLSN].Data.AfterImage;
        Cache[P].PageLSN = RedoLSN;
      else
        DirtyPages[P].RecLSN = Cache[P].PageLSN + 1;
      endif;
    endif;
  endif;
  RedoLSN = RedoLSN + 1;
RedoTerm [Internal]
Precondition:
  Redo ∈ ActiveSet;
  RedoLSN = LSN;
Effects:
  ActiveSet = {Undo1};
```

Fig. 18. Redo actions.

given in Section 3. Any proof enhances our confidence in an algorithm, but we believe that this article also makes a contribution to our understanding of the way the algorithm works. Our proof is built on a sequence of propositions, each of which can enhance the intuition about the algorithm by revealing important relationships (called *invariants*) that always hold among different aspects of the system state. We expect that development and proof of variant algorithms can be guided by the need to keep relationships like these.

The definition of correctness in Section 3 is stated in terms of the sequence of input and output actions produced by the data manager, because this is the way the rest of the system interacts with the manager. This definition is rather general. For example, it would apply to an algorithm that did not keep any copy of the current state of the database, but rather used the log as the sole source of information, scanning it on every read to find the appropriate value. An algorithm such as that would be quite easy to verify, but the performance would be unacceptable. To obtain fast response time and recovery time, a data manager needs to keep copies of some items in volatile cache, allowing data to move between cache and stable storage with few restrictions;[14] it also needs to periodically take *checkpoints* that place additional information in the log that can be used at

[14]This is usually described as a steal/no-force strategy of buffer management [Gray and Reuter 1993; Harder and Reuter 1983].

```
Undo1 [Internal]
Precondition:
  Let UndoingLSN = max({TransTable[T'].UndoNxtLSN:TransTable[T'] ≠ ∅ });
  Let x = Log[UndoingLSN].Data.DataItemName;
  Let T = Log[UndoingLSN].TransID;
  Undo1 ∈ ActiveSet;
  ∃ T' such that TransTable[T'] ≠ ∅;
  Cache[Page(x)] ≠ ∅;
Effects:
  ActiveSet = (ActiveSet - {Undo1}) ∪ {Undo2(UndoingLSN)};
  Log[LSN].Type = Comp;
  Log[LSN].TransID = T;
  Log[LSN].PrevLSN = TransTable[T].LastLSN;
  Log[LSN].UndoNxtLSN = Log[UndoingLSN].PrevLSN;
  Log[LSN].PageID = Page(x);
  Log[LSN].Data.DataItemName = x;
  Log[LSN].Data.AfterImage = Log[UndoingLSN].Data.BeforeImage;
  TransTable[T].UndoNxtLSN = Log[UndoingLSN].PrevLSN;
  TransTable[T].LastLSN = LSN;
  Cache[Page(x)].x = Log[UndoingLSN].Data.BeforeImage; /* Undo */
  Cache[Page(x)].PageLSN = LSN;
  LSN = LSN + 1;
Undo2(U) [Internal]
Precondition:
  Undo2(U) ∈ ActiveSet;
Effects:
  Let T = Log[U].TransID;
  ActiveSet = (ActiveSet - {Undo2(U)}) ∪ {Undo1};
  if (Log[U].PrevLSN = 0)
    Log[LSN].Type = End; Log[LSN].TransID = T;
    Log[LSN].PrevLSN = TransTable[T].LastLSN; Log[LSN].Data = Rollback;
    TransTable[T] = ∅;
  endif;
AckRestart [Output]
Precondition:
  Undo1 ∈ ActiveSet;
  ∀ transactions T, TransTable[T] = ∅;
Effects:
  ActiveSet = ∅;
  SystemState = Normal;
```

Fig. 19. Undo pass actions.

restart to limit the amount of the log that must be examined. The ARIES algorithm uses all these ideas, but the precise implementation depends on a range of complex data structures that must be processed in subtle ways. It is these aspects of the system that offer good performance, but they also complicate the verification.

The main data structures used in ARIES are the stable and cached copies of pages containing data items, the log, the transaction table, and the dirty pages table. Our proof is structured, with a sequence of key propositions that show relationships between the values of these data structures. During normal processing, the relationships are obvious. For example, the transaction table entry for an active transaction T has *LastLSN* field pointing to the last log record that concerns T. In order to prove a statement such as this, however, we need to strengthen it to give an

```
Log[0].Type = Update;
Log[0].TransID = $t_0$
Log[0].PrevLSN = 0;
Log[0].UndoNxtLSN = 0;
Log[0].Data.DataItemName = px;
Log[0].Data.BeforeImage = 0;
Log[0].Data.AfterImage = 0;
StableLSN = LSN = 1;
for all data items $x$,
  Stable[Page(x)].x = 0;
endfor;
for all transactions T,
  TransTable[T] = $\emptyset$;
endfor;
for all pages $P$
  Stable[P].PageLSN = 1;
  Cache[P] = $\emptyset$; DirtyPages[P] = $\emptyset$
endfor;
MasterRec = 1;
SystemState = Normal;
```

Fig. 20. Start state.

invariant relationship: one that holds in all states of the execution, including during restart. It is the main contribution of this article to identify the invariants. Once the correct invariant relationship is stated, it is proved by induction along the execution. That is, one can easily see that the relationship holds initially, and that from a state where the relationship holds, any possible action will result in another state satisfying the relationship. In some cases the proofs themselves depend on other results presented as lemmas. These are less meaningful in understanding the algorithm, and their proofs are shown in Kuo [1992b].

Here we give a brief summary of the four main propositions of this section. Proposition 5.2.5 gives the relationship between the transaction table and the log. In normal processing there is an entry in the transaction table associated with each transaction for which the log contains an *update* but not an *end* log record. Each entry contains two pointers that respectively point to the last log record written by the transaction and the next log record to be undone in the case of a rollback. The invariant that strengthens this statement is essentially concerned with the analysis pass of restart processing. Similarly, there is an invariant for the dirty pages table in Proposition 5.3.3. At the end of the analysis pass, the combination of the two invariants implicitly shows that both tables have been correctly reconstructed. Proposition 5.4.5 concerns the relationship between the values stored in each page and the log. During normal processing, this says that the effects of all the updates and compensations recorded in the log are reflected in the database. The invariant strengthens this, and implicitly shows the correctness of the redo pass. Finally, Proposition 5.5.5 shows the correctness of the undo pass and of transaction rollback. It states that when a transaction terminates, each data item that the transaction has accessed contains the value written by the last transaction that wrote to the item and committed.

Based on the propositions, we can complete the argument that the data manager is correct. This is done in Theorem 5.6.1. The essential observation is to connect properties of the state with properties of the sequence of inputs and outputs. For example, the values written in update operations are just the after images in update log records.

5.1 Basic Properties

The log is the central data structure used in the data manager. Here we introduce notation that we need later. We also state and prove some basic properties.

We use the following notation to express that an interval of the log is equivalent in two states S_i and S_j.

Notation 5.1. For two states S_i and S_j of the data manager, we write $S_i.Log[a...b] = S_j.Log[a'...b']$ if and only if $a = a'$, $b = b'$ and $\forall l, a \leq l \leq b$, $S_i.Log[l] = S_j.Log[l]$.

The state of the data structures such as the transaction table, dirty pages table, pages, and the data items in the DBMS are all related to the state of the log. The invariants that express these relationships are all proved by induction. For the proofs, we need to know conditions when the state of parts of the log does not change. Volatile storage is corrupted when a crash occurs since it causes all the log records in volatile storage to be lost (change). In the following lemma, we show that if a log record is unchanged between state S_i to S_j, then all the preceding log records are also unchanged between the two states. The corollary, which follows, then shows that if an action π_i writes a log record $S_i.Log[l]$ that remains unchanged in some later state S_j, then all the log records that precede $S_i.Log[l]$ in state S_i are also unchanged.

LEMMA 5.1.2 *Suppose i, j, l are such that* $0 \leq i < j$, $0 \leq l < S_i.LSN$, *and* $\forall k$, $(i \leq k \leq j) \Rightarrow (S_i.Log[l] = S_k.Log[l] \neq \emptyset)$. *Then* $S_i.Log[1..l] = S_j.Log[1..l]$.

PROOF. The proof is straightforward as the only action that can modify a nonempty log record is the action *Crash*. □

COROLLARY 5.1.3. *Suppose i, j, l are such that* $i < j$, $S_{i-1}.Log[l] = \emptyset$, $S_i.Log[l] \neq \emptyset$, *and* $\forall k$, $i \leq k \leq j$, $S_k.Log[l] = S_i.Log[l]$. *Then* $S_i.Log[1..l] = S_j.Log[1..l]$.

PROOF. The proof follows immediately from Lemma 5.1.2. □

Whenever a read or write to a data item is requested, the page containing the data item is first fetched into cache if it is not already there. In the case of a write, the new value is first recorded in the cache and, at some later stage, flushed to stable storage. In Definition 5.1.4, we introduce notation to represent the effective state of the database for a page. This is written as if it were a state component; when there is a copy of the page in volatile cache, the value of that copy is the effective state of the page; when the page is not in the cache, the effective state is the value in stable storage. The effective state abstracts away the fact that there is volatile and stable storage, and allows us to refer to the current state of the database easily in later proofs.

Definition 5.1.4. For each page P in the database, define

$$S_n.DB[P] = \begin{cases} S_n.Cache[P] & \text{if } Cache[P] \neq \emptyset \\ S_n.Stable[P] & \text{otherwise.} \end{cases}$$

The *PageLSN* in each page points to the last log record that describes the latest operation to the page. Because accesses (reads and writes) are through the cache, a page in stable storage may not contain the effects of all the operations (the effects of some updates may be in cache and have not been flushed to stable storage). Therefore, for any page P in any state S_n,

$$S_n.DB[P].PageLSN \geq S_n.Stable[P].PageLSN.$$

Also, the effects of an operation that has been flushed to stable storage is never lost. Therefore, for each page in stable storage, the *PageLSN* is monotonic increasing. These two basic properties are stated in the following lemmas.

LEMMA 5.1.5 *Let P be any page, and S_n any state. Then*

$$S_n.Stable[P].PageLSN \leq S_n.DB[P].PageLSN.$$

PROOF. The crucial step of the induction proof is to realize that whenever the page is modified in volatile storage, either in an update or in a compensation, the page's *PageLSN* increases. It is then a straightforward induction proof. □

LEMMA 5.1.6 *For any i such that $0 \leq i < n$ and for any page P,*

$$S_i.Stable[P].PageLSN \leq S_n.Stable[P].PageLSN.$$

PROOF. The proof is by induction and it follows from Lemma 5.1.5 that the only action that can modify a page P in stable storage is *Flush*(P). □

Checkpoints collate information on the state of the database that is later used by the analysis pass of recovery. In the final lemma in this section, we state some basic properties of checkpoints that are used in the later lemmas and propositions. For each *end checkpoint* log record, there must have been a checkpoint action that wrote the log record. Also, there is an action that wrote the corresponding *begin checkpoint* log record. Each transaction's entry in the checkpoint's copy of the transaction table is a copy of the transaction's entry in the transaction table sometime during the checkpoint (we do not know the exact state as checkpoints are taken asynchronously). A similar property is true of the dirty pages table.

LEMMA 5.1.7 *Suppose $S_n.Log[l].Type = EndCheckpt(BLSN)$. Then there are b, e, such that $1 \leq b < e \leq n$ and the following hold.*

(1) $\pi_b = BgnCheckpt$, $\pi_e = EndCheckpt(S_b.LSN - 1)$;
(2) $\forall k,\ b < k < e,\ \pi_k \notin \{BgnCheckpt, Crash\}$;

(3) $\forall k,\ e \leq k \leq n,\ S_k.Log[S_{e-1}.LSN] = S_n.Log[l]$;

(4) for each transaction T, if $S_n.Log[l].Data.TransTable[T] \neq \emptyset$, *then* $\exists i$ *such that* $b < i < e$, $S_i.TransTable[T] = S_n.Log[l].Data.TransTable[T]$ *else* $S_e.TransTable[T] = \emptyset$;

(5) for each page P, if $S_n.Log[l].Data.DirtyPages[P] \neq \emptyset$, *then* $\exists j$ *such that* $b < j < e$ *and* $S_j.DirtyPages[P] = S_n.Log[l].Data.DirtyPages[P]$ *else* $S_e.DirtyPages[P] = \emptyset$.

PROOF. The proof is straightforward by inspection of the model. □

5.2 Transaction Table

We now present the first major invariant that states precisely what is contained in the transaction table during recovery and normal processing. We say that a transaction is active during normal processing if the log contains a record that is written by the transaction, and the log does not contain its *end* log record (i.e., the transaction has not committed or rolled back).[15] The invariant strengthens the observation that during normal processing a transaction has an entry in the transaction table if and only if the transaction is active. Also, each entry contains two pointers: the first points to the last log record written by the transaction (*LastLSN*) and the other points to the next log record to be undone in the case of a rollback (*UndoNxtLSN*).

After a crash, the contents of the transaction table are lost and during the analysis pass, the table is reconstructed. A trivial algorithm to reconstruct the table is to scan the complete log. If such an algorithm were used, there would be a simple invariant: a transaction has an entry in the transaction table if and only if the portion of the log that the pass has processed contains a record written by the transaction and does not contain its *end* log record. Scanning the complete log after a failure is obviously too inefficient, which is why checkpoints are required. Processing checkpoint information stored in an *end checkpoint* log record has the same net effect as scanning all the log records that precede the *begin checkpoint* log record. Informally, the invariant for the transaction table is as follows.

(1) During normal processing, redo pass, and undo pass of recovery, the transaction table reflects the state of the whole log.
(2) During the analysis pass,
    (a) if it has not processed an *end checkpoint* log record, then the transaction table reflects the state of all the log records the analysis pass has processed;
    (b) otherwise, the table reflects all the log records from the beginning of the log up to the log record it is currently processing.

[15]A transaction that has not made any updates needs no undo or redo in a failure which is why we say a transaction is active only after it has made an update.

Before we formally state and prove the invariant on the transaction table we present four definitions. They identify the lower bound (S_n.*LowerBound*) and the upper bound (S_n.*UpperBound*) of the portion of the log that the transaction table reflects. During normal processing, the lower bound points to the beginning of the log and the upper bound points to the end of the log, for example. The set S_n.*LogBy*(T, i, j) is defined as the set of log sequence numbers whose corresponding log record lies in the interval $[i, j]$ of the log and are written by the transaction T. We then define S_n.*LastLogBy*(T, i, j) to be the greatest log sequence number in S_n.*LogBy*(T, i, j). A transaction has an entry in the transaction table if the set S_n.*LogBy*(T, S_n.*LowerBound*, S_n.*UpperBound*) is not empty and the transaction's entry in the transaction table is dependent on the type of the log record pointed to by S_n.*LastLogBy*(T, S_n.*LowerBound*, S_n.*UpperBound*).

Definition 5.2.1. Let S_n be any state.

$S_n.LowerBound =$

$$\begin{cases} S_n.MasterRec & \text{if } Analysis \in ActiveSet \quad \text{and} \quad S_n.FirstCheckpt = True \\ 1 & \text{otherwise.} \end{cases}$$

Definition 5.2.2. Let S_n be any state.

$$S_n.UpperBound = \begin{cases} S_n.AnalysisLSN - 1 & \text{if } Analysis \in S_n.ActiveSet \\ S_n.LSN - 1 & \text{otherwise.} \end{cases}$$

Definition 5.2.3. Let S_n be any state.

$S_n.LogBy(T, i, j) =$

$$\{l : i \le l < j, S_n.Log[l].Type = Update|Comp|End, S_n.Log[l].TransID = T\}.$$

Definition 5.2.4.

$$S_n.LastLogBy(T, i, j) = max(\{l : l \in S_n.LogBy(T, i, j)\} \cup \{0\}).$$

Using the notation previously defined, we now state and prove the first important invariant of the data manager.

PROPOSITION 5.2.5 *For any transaction T, let* $S_n.LLB = S_n.LastLogBy(T, S_n.LowerBound, S_n.UpperBound)$.

If $S_n.SystemState \neq Down$, $S_n.LogBy(T, S_n.LowerBound, S_n.UpperBound) \neq \emptyset$, *and there is no* i *such that* $S_n.LowerBound \le i \le S_n.UpperBound$, $S_n.Log[i].Type = End$, $S_n.Log[i].TransID = T$, *then*

(1) $S_n.TransTable[T].LastLSN = S_n.LLB$ *and*

(2) $S_n.TransTable[T].UndoNxtLSN =$

$$\begin{cases} S_n.Log[S_n.LLB].UndoNxtLSN & if\ S_n.Log[S_n.LLB].Type = Comp \\ S_n.LLB & otherwise; \end{cases}$$

otherwise, $S_n.TransTable[T] = \emptyset$.

PROOF. The proof is by induction. During normal processing, redo pass, and undo pass the proof is straightforward as a transaction related log record is written by an action if and only if the action also updates the transaction table appropriately. During the analysis pass, if the pass is processing the last successful checkpoint's log record, the proof shows the correctness of the checkpointing procedure (gathers the right information) and of the analysis pass processing the checkpoint information; otherwise, the proof is similar to the proof during normal processing as when the pass processes a transaction related log record, the transaction table is updated appropriately. The proof can be found in Appendix B. □

5.3 Dirty Pages Table

We now show the invariant that states the correctness of the dirty pages table. It is very similar to the invariant for the transaction table; however, there is one crucial difference. In the transaction table, the entries have to be *exact*—a transaction has an entry in the table if and only if the transaction is active and each pointer in each entry can point to exactly one location in the log (there is no flexibility). This is not the case for the dirty pages table.

A page may have an entry in the dirty pages table even though it is not dirty. Also, the recovery *LSN* does not need to point to a particular location in the log. In fact, the algorithm is still correct even if all pages have an entry in the table and the recovery *LSN* points to the beginning of the log. This would be very inefficient as the redo pass will then need to process the whole log. However, it would be correct, and in fact our proof works as long as the dirty pages table satisfies quite weak conditions.

The dirty pages table is correct as long as each dirty page has an entry in the dirty pages table and the recovery *LSN*(*RecLSN*) points to somewhere in the log such that it partitions the log into two portions: in the front portion ($S.Log[1..S.DirtyPages[P].RecLSN]$) the effects of all recorded operations for the page P are reflected in stable storage; however, in the back portion this may not be the case. The reason for the flexibility is that the dirty pages table and the log only indicate which recorded operations are *potentially* missing from the database. The redo pass compares the log sequence number of the log record and the *PageLSN* of the page that the recorded operation affected to determine if the effects of the operation are really missing from the database and thus requiring redo.

We define the set $S_n.Unflush(P, LowerBound, UpperBound)$ to be the set of recorded operations in the interval [*LowerBound, UpperBound*] of the log that are potentially missing from the page P. We then define the set $S_n.ReallyUnflush(P)$ to be the set of updates that are actually missing from the database.

Definition 5.3.1

$$S_n.Unflush(P, i, j) = \left\{ l : \begin{array}{l} i \le l \le j,\ S_n.Log[l].PageID = P, \\ \nexists e,\ l < e < j,\ S_n.Log[e].Type = OsFileReturn,\ S_n.Log[e].PageID = P \end{array} \right\}.$$

Definition 5.3.2

$$S_n.ReallyUnflush(P) = \left\{ l : \begin{array}{l} l \in S_n.Unflush(P, 1, S_n.LSN - 1) \\ \text{and } l > S_n.Stable[P].PageLSN \end{array} \right\} \cup \{S_n.LSN\}.$$

The invariant for the dirty pages table is presented in the following proposition.

PROPOSITION 5.3.3 *Let P be a page in the database. If $S_n.SystemState \ne Down$ and $S_n.Unflush(P, S_n.LowerBound, S_n.UpperBound) \ne \emptyset$, then $S_n.DirtyPages[P].RecLSN < min(S_n.ReallyUnflush(P))$ else $S_n.DirtyPages[P] = \emptyset$.*

PROOF. The proof is similar to the proof of Proposition 5.2.5 as the dirty pages table and transaction table have similar functionality. The dirty pages table keeps track of dirty pages whereas the transaction table keeps track of transactions. Also, the analysis pass uses similar procedures to reconstruct the tables. □

For each dirty page in the database, we have found an upper bound for its entry in the dirty pages table. In fact a lower bound also holds.

In state S_n, if $S_n.Unflush(P, S_n.LowerBound, S_n.UpperBound - 1) \ne \emptyset$, then

$$S_n.DirtyPages[P].RecLSN \ge min(S_n.Unflush(P, S_n.LowerBound, S_n.UpperBound - 1).$$

The lower bound shows us the maximum number of log records that the redo pass has to process.

What we have implicitly shown in Propositions 5.2.5 and 5.3.3 and from the definitions of $S_n.LowerBound$ and $S_n.UpperBound$ is the correctness of the analysis pass and the checkpointing procedure.

5.4 The Page

During normal processing the value stored in each data item is consistent with what is recorded in the log; that is, the value stored in each data item is the after image of the last log record that recorded an operation to the data item. After a crash, this relationship is no longer valid as the effects of some operations are lost. The redo pass re-establishes the relationship by redoing the effects of the missing operations. Then the relationship continues to hold during the undo pass. Proposition 5.4.5 formally describes the relationship during the undo pass and normal processing. It also describes the relationship during the redo pass—how the redo pass restores the effects of missing operations.

The proposition is in parts:

(1) During the redo pass, it states that for each data item, the value stored in the data item is the *after image* of the log record that describes the latest *update* or *compensation* operation to the data item in some interval $[1..l]$[16] *of the log where* $RedoLSN - 1 \leq l \leq LSN - 1$.
(2) During normal processing and the undo pass, the invariant is the same except $l = LSN - 1$; that is, the value stored in each data item is the *after image* of the latest log record that recorded an operation to the data item. Notice that at the end of the redo pass, $RedoLSN = LSN$ so the previous invariant becomes identical to this one.

In the proof of the proposition, we show that whenever the redo pass processes a log record, it redoes the operation if the effects are missing. The proof needs to identify the state of each data item, which is formally stated in Lemmas 5.4.3 and 5.4.4, as the proof needs to show that a recorded operation is redone if and only if the effects are missing. In Lemma 5.4.3 we show that during the redo pass, each data item contains the effects of all operations whose corresponding log record lies in the interval $[1..RedoLSN - 1]$ of the log. Then, in Lemma 5.4.4, we show that if a page is not dirty, then the effects of all recorded operations that affected the page are not missing.

The lemmas and proposition need to identify the *LSN* of the last log record in the interval $[a..b]$ of the log, which recorded an operation to a data item x, also, the *PageLSN* of the page where the data item x resides. Notations for these concepts are defined in the following.

Definition 5.4.1

$S_n.Lwr(a, b, x) =$

$$max\left(\left\{i: a \leq i \leq b, \begin{array}{l} x = S_n.Log[i].Data.DataItemName \\ S_n.Log[i].Type = Update|Comp \end{array}\right\} \cup \{0\}\right).$$

Definition 5.4.2

$$S_n.PgLSN(x) = S_n.DB[Page(x)].PageLSN.$$

LEMMA 5.4.3 *Suppose* $S_n.SystemState \neq Down$ *and* $Analysis \notin S_n.ActiveSet$. *For each data item* x

[16]The invariant actually states something a little stronger by precisely stating the value of l.

(1) $S_n.DB[Page(x)].x = S_n.Log[S_n.Lwr(1, S_n.PgLSN(x), x)].Data.AfterImage;$

(2) *if Redo* $\in$ S_n*.ActiveSet then* $\forall l$, $S_n.PgLSN(x) < l < S_n.RedoLSN$, *either*

$$S_n.Log[l].PageID \neq Page(x) \text{ or } S_n.Log[l].Type \neq Update|Comp$$

(3) *if Redo* $\notin$ S_n.ActiveSet then $\forall l$, $S_n.PgLSN(x) < l < S_n.LSN$, *either*

$$S_n.Log[l].PageID \neq Page(x) \text{ or } S_n.Log[l].Type \neq Update|Comp.$$

PROOF. The proof is shown in Kuo [1992b]. □

LEMMA 5.4.4 *For any data item x, let*

$$S_n.LastFlushLSN(x) =$$

$$max\left(\left\{i: \begin{array}{l} 1 \leq i < S_k.LSN, S_n.Log[i].Type = OsFileReturn, \\ S_n.Log[i].PageID = Page(x) \end{array}\right\} \cup \{0\}\right).$$

If $S_n.Unflush(Page(x), 1, S_n.LSN - 1) = \emptyset$, *then*

$$S_n.DB[Page(x)].x$$

$$= S_n.Log[S_n.Lwr(1, S_n.LastFlushLSN(x), x)].Data.AfterImage$$

$$= S_n.Log[S_n.Lwr(1, S_n.LSN - 1, x)].Data.AfterImage.$$

PROOF. The proof is a straightforward induction proof and is shown in Kuo [1992b]. □

PROPOSITION 5.4.5 *For any page P, let* $S_n.MaxLSN(P)=$

$$max(S_n.RedoLSN - 1, S_n.DirtyPages[P].RecLSN - 1, S_n.DB[P].PageLSN)$$

$$S_n.UpdateToLSN(P) =$$

$$\begin{cases} S_n.LSN - 1 & \text{if } S_n.Unflush(P, 1, S_n.LSN-1)=\emptyset \\ S_n.MaxLSN(P) & \text{otherwise.} \end{cases}$$

If Redo $\in S_n.ActiveSet$, *then*

$S_n.DB[Page(x)].x =$

$$S_n.Log[S_n.Lwr(1, S_n.UpdateToLSN(P), x)].Data.AfterImage,$$

else if Undo $\in S_n.ActiveSet$ *or* $S_n.SystemState = Normal$*, then*

$$S_n.DB[Page(x)].x = S_n.Log[S_n.Lwr(1, S_n.LSN, x)].Data.AfterImage.$$

PROOF. The proof is by induction and is shown in Appendix C. □

5.5 The Database

The final proposition states precisely what value is stored in a data item when a transaction that updated the data item terminates; that is, the value stored in the item is the last committed value (the value written by the last transaction that updated the data item *and committed*). This is crucial to prove correctness, because the last committed value is what should be returned by a subsequent read of that item. This last step to prove the correctness of the data manager is shown in Section 5.6.

ARIES uses an update-in-place policy; that is, each data item has only one slot in the database.[17] When a transaction makes an update to a data item, it updates the only copy. Thus if the transaction later rolls back due to a crash or transaction failure, the update *must* be undone. In Lemma 5.5.4 we show that whenever an *update* is undone, the data manager restores the last committed value. When a transaction completes its rollback—all updates have been rolled back—the last committed value is restored in each of the data items updated by the transaction. An *update* is undone by copying the before image from the log record to the data item. Therefore we must show that the before image is the current last committed value for the data item. In Lemma 5.5.3 we first show that the before image of an *update* log record is the after image of the previous log record that recorded an operation to the data item. Because the execution is strict, the after image is the data item's current last committed value. When a transaction commits, it is elementary to show from the definitions that each data item that was updated by the transaction contains the last committed value.

Lemmas 5.5.3 and 5.5.4 and Proposition 5.5.5 need to identify the log record that recorded the last *update* to the data item by a committed transaction.

Notation for this concept, the last committed write, is defined in the following.

[17]In contrast, with methods such as shadow paging or multiversion techniques, each item has several locations.

Definition 5.5.1

$$S_n.Lcw(a, b, x) = max\left(\left\{i: \begin{array}{l} a \le i \le b, x = S_n.Log[i].Data.DataItemName, \\ S_n.Log[i].TransID \in S_n.CommittedTrans. \end{array}\right\} \cup \{0\}\right)$$

where $S_n.CommittedTrans =$

$$\left\{T: \exists j, \begin{array}{l} S_n.Log[j].Type = End, S_n.Log[j].TransID = T, \\ S_n.Log[j].Data = Commit \end{array}\right\}.$$

Also, we need to identify when an *update* has been undone. We define *UpdateLSN*(T, x) to be the log sequence number of the log record that recorded an *update* to a data item x by a transaction T. Lemma 5.5.3 shows that when *UpdateLSN*(T, x) is greater than T's *UndoNxtLSN* entry in the transaction table, then the update recorded in the log record pointed to by *UpdateLSN*(T, x) has been undone and the last committed value is restored in the data item.

Definition 5.5.2

$$S_n.UpdateLSN(T, x) = \begin{cases} i & \text{if } \begin{array}{l} \exists i\ S_n.Log[i].Type = Update, S_n.Log[i].TransID = T, \\ S_n.Log[i].Data.DataItemName = x, \end{array} \\ -1 & \text{otherwise.} \end{cases}$$

Because each transaction can update a data item at most once, the value is well defined.

LEMMA 5.5.3 *If $S_n.Log[l].Type = Update$ and $S_n.Log[l].Data.DataItemName = x$, then*

$$S_n.Log[l].Data.BeforeImage = S_n.Log[S_n.Lwr(1, l-1, x)].Data.AfterImage.$$

PROOF. Proof follows from Proposition 5.4.5. □

LEMMA 5.5.4 *Suppose $S_n.SystemState = Normal$ or $Undo \in S_n.ActiveSet$. If $S_n.TransTable[T] \neq \emptyset$ and $\exists i, v$ such that $\pi_i = ReqWrite(T, x, v)$ then*

$$S_n.DB[Page(x)].x =$$

$$\begin{cases} S_n.Log[S_n.Lwr(1, S_n.LSN - 1, x)]. & if\ S_n.Undone(T, x) = False \\ & Data.AfterImage \\ S_n.Log[S_n.Lcw(1, S_n.LSN - 1, x)]. & otherwise \\ & Data.AfterImage \end{cases}$$

where $S_n.Undone(T, x) =$

$$\begin{cases} True & if\, S_n.UpdateLSN(T, x) < S_n.TransTable[T].UndoNxtLSN \\ False & otherwise. \end{cases}$$

PROOF. The proof is by induction and shows that whenever an *update* to a data item is undone, the data item's last committed value is restored. The proof is shown in Kuo [1992b]. □

We now present the proposition that shows that when a transaction terminates, the correct value is stored in each of the data items that the transaction updated.

PROPOSITION 5.5.5 *Suppose* $S_n.SystemState = Normal$. *If* $\exists T, i, v$ *such that* $0 \le i \le n$, $T \in S_n.TermTrans$, $\pi_i = ReqWrite(T, x, v)$ *and* $\forall j, v', i < j \le n$, $\pi_j \ne ReqWrite(T, x, v)$, *then*

$$S_n.DB[Page(x)].x = S_n.Log[S_n.Lcw(1, S_n.LSN - 1, x)].Data.AfterImage.$$

PROOF. The proof is by induction and it shows that whenever a transaction terminates, each data item that the transaction updated contains the last committed value, no matter if the transaction committed or rolled back. The proof is shown in Appendix D. □

5.6 Correctness

It is now straightforward to show that the DBMS is correct. The scheduler provides strict schedules, so whenever the data manager receives a read request from the scheduler, it knows that the previous transaction that wrote to the item has terminated (unless the reader transaction itself wrote the item). For example, conventional two-phase locking is strict, because the reader's shared lock would conflict with the exclusive lock held by a different, unterminated, writer. Because the read observes the effective state of the item, and Proposition 5.5.5 shows that this value is the one written by the last committed writer, we find that each read will return the appropriate value.

THEOREM 5.6.1 *Let* $\alpha = S_0\pi_1S_1\pi_2S_2\pi_3S_3, \ldots, \pi_nS_n$ *be any allowable execution of the data manager that is well formed and strict on data items. Then* α *is correct, as defined in Definition* 3.2.6.

PROOF. The proof is presented in Appendix E. □

This completes the proof of the recovery algorithm based on ARIES and presented in Section 4.

LSN	Type	TransID	···	Data
1	Update	T_1		x, $BeforeImage_x$, $AfterImage_x$
2	Update	T_2		y, $BeforeImage_y$, $AfterImage_y$
3	BgnCheckpt			
4	Update	T_1		z, $BeforeImage_z$, $AfterImage_z$
5	End	T_1		Commit
6	EndCheckpt			$TransTable$, $DirtyPages$

Fig. 21. State of the stable log after the crash.

6. CONCLUSION AND FUTURE WORK

In this article we have used a formal method for specification, modeling, and verification, and applied this method to ARIES which is an important recovery algorithm. The contributions of this article are a definition of correctness that can be used with many different recovery algorithms, a detailed mathematical model of a recovery algorithm that includes the main features of ARIES, a proof that shows that the model has the desired property, and several intermediate results that show important relationships among data structures during execution of the algorithm.

We have already applied these techniques to other recovery algorithms [Kuo and Fekete 1994]. In our future work, we would like to extend our model and proof to include logical logging and partial rollbacks. We also will explore combining this with existing proofs of concurrency control, to show the correctness of a complete transaction processing system. In addition, we hope to incorporate distribution and a commit protocol.

APPENDIX

A. EXAMPLE

Suppose a crash has just occurred and the state of the log is shown in Figure 21. Also, suppose the checkpoint took a copy of T_1's entry in the transaction table after T_1 updated the data item z but before it committed. Thus the transaction T_1 will have an entry in the checkpoint's copy of the transaction table.

The restart procedure's first pass is the analysis pass. The pass will start processing from the *begin checkpoint* log record. It will then insert an entry into the transaction table for the transaction T_1 when it processes the second *update* log record written by T_1 (log record whose *LSN* is 4). Then the pass will delete the entry from the transaction when it processes T_1's *end commit* log record. When the analysis pass processes the *end checkpoint* log record, it should insert an entry into the transaction table for the transaction T_2 as T_2 was active at the time of the crash. However, it *should not* insert an entry for T_1 into the transaction table as T_1 has terminated (committed in this case). If an entry for T_1 is inserted into the transaction table, then the undo pass (the final pass of recovery) will undo the updates by T_1 as the pass will undo all updates from transactions who have an entry in the reconstructed transaction table, thus undoing the effects of updates from committed transactions. Hence under our assumption that the latch on a transaction's entry is released after the checkpoint has taken a copy of the entry, it is crucial to make the additional check that the transaction did not terminate (commit or abort) during the checkpoint.

B. Proof of Proposition 5.2.5

It is straightforward to show that the statement is true in state S_0.

Assume that the statement is true in all states S_i such that $i \leq k$. To complete the induction, we must show that the statement is true in state S_{k+1}. The nontrivial cases are those actions π_{k+1} that modify the transaction table and/or the log. By inspection of the model, the cases are:

$$\pi_{k+1} \in \left\{ \begin{matrix} \mathit{Write}(T, x, v), \mathit{Commit}(T), \mathit{Rollbk}(T), \mathit{RollbkTerm}(T), \\ \mathit{Crash}, \mathit{Analysis}, \mathit{AnalysisTerm}, \mathit{Undo}1, \mathit{Undo}2(U) \end{matrix} \right\}.$$

Case 1. $\pi_{k+1} \in \{\mathit{Write}(T, x, v), \mathit{Rollbk}(T), \mathit{Undo}1\}$. By inspection of the action π_{k+1} and from the induction hypothesis, it is elementary to show that the statement is true in state S_{k+1}.

Case 2. $\pi_{k+1} \in \{\mathit{Commit}(T), \mathit{RollbkTerm}(T), \mathit{AnalysisTerm}, \mathit{Undo}2(U)\}$. *The action* π_{k+1} writes a transactions *end* log record to the log if and only if it removes the transaction's entry from the transaction table. Hence, from the induction hypothesis, the statement is true in state S_{k+1}.

Case 3. $\pi_{k+1} = \mathit{Analysis}$. The proof is split into two parts. In this first case, the analysis pass is not processing the last successful checkpoint's *end* log record whereas in the later case it is.

Case 3.1. $S_k.\mathit{Log}[S_k.\mathit{AnalysisLSN}].\mathit{Type} \neq \mathit{EndCheckpt}$ or $S_k.\mathit{FirstCheckpt} = \mathit{False}$. It is elementary to show that

$$S_{k+1}.\mathit{LowerBound} = S_k.\mathit{LowerBound} \text{ and } S_{k+1}.\mathit{UpperBound}$$
$$= S_k.\mathit{UpperBound} + 1.$$

The upper bound is incremented by one and if $S_k.\mathit{Log}[S_k.\mathit{AnalysisLSN}]$ is a transaction related log record, then the transaction table is updated in the same way as if the log record were written during normal processing (similar to Case 1). If the log record is not transaction related, the action has no effect on the transaction table or the log.

It follows from the induction hypothesis and the effects of the action that the statement is true in state S_{k+1}.

Case 3.2. $S_k.\mathit{Log}[S_k.\mathit{AnalysisLSN}].\mathit{Type} = \mathit{EndCheckpt}$ and $S_k.\mathit{FirstCheckpt} = \mathit{True}$. *Let* T be a transaction. The proof for the transaction T is in two cases.

Case 3.2.1 $S_k.\mathit{LogBy}(T, S_k.\mathit{LowerBound}, S_k.\mathit{UpperBound}) \neq \emptyset$. In this case, the analysis pass has already processed a log record from the transaction T. The action π_{k+1} in this case does not modify T's entry in the transaction table as it is already correct. We now formally prove that the statement is true in state S_{k+1}. From the definitions, $S_k.MasterRec = S_{k+1}.LowerBound = 1$. It is elementary to show that the log sequence number stored in the master record is monotonic increasing. Therefore, $S_{k+1}.LowerBound = 1 \le S_k.MasterRec = S_k.LowerBound$, also,

$$S_{k+1}.UpperBound = S_k.UpperBound + 1$$

$$S_{k+1}.Log[S_{k+1}.UpperBound].Type \neq Update|Comp|End$$

Hence

$$S_{k+1}.LogBy(T, S_{k+1}.LowerBound, S_{k+1}.UpperBound)$$

$$\supseteq S_k.LogBy(T, S_k.LowerBound, S_k.UpperBound)$$

Therefore

$$S_{k+1}.LastLogBy(T, S_{k+1}.LowerBound, S_{k+1}.UpperBound)$$

$$= S_k.LastLogBy(T, S_k.LowerBound, S_k.UpperBound).$$

From the effects of the action π_{k+1},

$$S_{k+1}.TransTable[T] = S_k.TransTable[T]$$

$$S_{k+1}.Log[1..S_{k+1}.LSN] = S_k.Log[1..S_k.LSN].$$

Therefore, from the induction hypothesis, the statement is true in state S_{k+1}.

Case 3.2.2 $S_k.LogBy(T, S_k.LowerBound, S_k.UpperBound) = \emptyset$. In this case, the transaction did not write any log records during the checkpoint (from the time it wrote the *begin checkpoint* log record till it wrote the *end checkpoint* log record). Thus *T*'s entry in the transaction table is unchanged during this period. Checkpoints are taken asynchronously and a copy of each nonempty entry in the table is taken sometime during the checkpoint. Hence the checkpoint's copy of *T*'s entry in the transaction table is the same as *T*'s entry in the transaction table at the end of the checkpoint. From the induction hypothesis, it follows that the statement is true in state S_{k+1}. We formally prove this below.

From the definitions, $S_k.LowerBound = S_k.MasterRec$. The two crucial steps of the proof are to show

$$S_{k+1}.TransTable[T] = S_k.Log[S_k.AnalysisLSN].Data.TransTable[T]$$

$$= S_e.TransTable[T]$$

$$S_k.Log[1..S_k.AnalysisLSN] = S_e.Log[1..S_e.LSN - 1],$$

where π_e was the action that wrote the *end checkpoint* log record. Then, by the induction hypothesis on state S_e and the effects of the action, it follows that the statement is true in state S_{k+1}.
From Lemma 5.1.7, there are b, e such that $b < e$,

$$\pi_b = BgnCheckpt,\ \pi_e = EndCheckpt(S_b.LSN - 1),$$

$$\forall k,\ b < k < e,\ \pi_k \notin \{BgnCheckpt,\ Crash\},$$

$$\forall k,\ e \le k \le n,\ S_k.Log[S_{e-1}.LSN] = S_n.Log[l],$$

and for each transaction T, if $S_n.Log[l].Data.TransTable[T] \neq \emptyset$, then

$$\exists \text{i such that } b < i < e,\ S_i.TransTable[T] = S_n.Log[l].Data.TransTable[T],$$

else $S_e.TransTable[T] = \emptyset$. It follows from Lemma 5.1.2

$$\forall j,\ b \le j \le e,\ S_e.Log[1 \ldots S_j.LSN - 1] = S_j.Log[1 \ldots S_j.LSN - 1] \text{ and}$$

$$S_{k+1}.Log[1 \ldots S_k.AnalysisLSN - 1] = S_e.Log[1 \ldots S_e.LSN - 1]. \qquad (1)$$

As $S_k.LogBy(T, S_k.LowerBound, S_k.UpperBound) = \emptyset$, it follows that $\forall b \le j \le e$,

$$S_e.LastLogBy(T, S_e.LowerBound, S_e.UpperBound) = S_j.LastLogBy(T, S_j.LowerBound, S_j.UpperBound).$$

Hence by the induction hypothesis, on the state S_e and S_i,

$$S_k.Log[S_k.AnalysisLSN].DataTransTable[T] = S_i.TransTable[T] = S_e.TransTable[T].$$

Therefore from the effects of the action π_{k+1},

$$S_{k+1}.TransTable[T] = S_k.Log[S_k.AnalysisLSN].Data.TransTable[T] = S_e.TransTable[T]. \qquad (2)$$

Hence, by (1), (2), and the induction hypothesis, the statement is true in state S_{k+1}. □

C. PROOF OF PROPOSITION 5.4.5

During the undo pass and normal processing, the proof is straightforward as the value stored in a data item is changed if and only if a log record is written to record the change. During the redo pass, the induction step (S', π, S) shows that if $S.MaxLSN(P) = S.RedoLSN - 1$, then the operation recorded in $S.Log[S.RedoLSN - 1]$ is redone if the effects are missing.

It is straightforward to show that the statement is true in state S_0.

Assume that the statement is true for all states S_i such that $i \leq k$. To complete the induction, we must show that the statement is true in state S_{k+1}.

Case 1. $\pi_{k+1} = AnaTerm$. Let x be any date item. There are two possible cases.

Case 1.1. $S_{k+1}.Unflush(Page(x), 1, S_{k+1}.LSN - 1) = \emptyset$. That is, the page is not dirty. Therefore, from Lemma 5.4.4,

$$S_{k+1}.DB[Page(x)].x =$$

$$S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].Data.AfterImage.$$

Hence the statement is true in state S_{k+1}.

Case 1.2. $S_{k+1}.Unflush(Page(x), 1, S_{k+1}.LSN - 1) \neq \emptyset$. That is, the page is dirty. The action sets the *RedoLSN* to be the minimum recovery *LSN* in the dirty pages table. That is,

$$S_{k+1}.RedoLSN = min(\{S_k.DirtyPages[P].RecLSN : S_k.DirtyPages[P] \neq \emptyset\}).$$

Therefore

$$S_{k+1}.UpdateToLSN(Page(x)) =$$

$$max(S_{k+1}.DirtyPages[Page(x)].RecLSN - 1, S_{k+1}.DB[Page(x)].PageLSN).$$

If $S_{k+1}.MaxLSN(Page(x)) = S_{k+1}.DB[Page(x)].PageLSN$, then it follows from Lemma 5.4.3 that the statement is true in the state S_{k+1}.

Suppose $S_{k+1}.UpdateToLSN(Page(x)) = S_{k+1}.DirtyPages[Page(x)].RecLSN$. If $\exists i$ such that

$$S_{k+1}.DB[Page(x)].PageLSN < i < S_{k+1}.DirtyPages[Page(x)].RecLSN,$$

$$S_{k+1}.Log[i].Type = Update|Comp \quad \text{and} \quad S_{k+1}.Log[i].PageID = Page(x),$$

then $i \in S_{k+1}.ReallyUnflush(Page(x))$. From Proposition 5.3.3, it follows that

$$S_{k+1}.DirtyPages[Page(x)].RecLSN \leq i.$$

This is a contradiction. Therefore, $\forall j$, $S_{k+1}.DB[Page(x)].PageLSN < j < S_{k+1}.DirtyPages[Page(x)].RecLSN$ $S_{k+1}.Log[i].Type \neq Update|Comp$ or $S_{k+1}.Log[i].PageID \neq Page(x)$. Hence it follows from Lemma 5.4.3 that the statement is true in state S_{k+1}.

Case 2. $\pi_{k+1} = Redo$. *If* $S_k.Log[S_k.RedoLSN].Type \neq Update \mid Comp$ then it is straightforward to show that the statement is true in state S_{k+1}.

Suppose $S_k.Log[S_k.RedoLSN].Type = Update \mid Comp$. Let $x = S_k.Log[S_k.RedoLSN].Data.DataItemName$. There are four possible cases for x. For any data item x' such that $x' \neq x$ but $Page(x') = Page(x)$, it follows from Lemma 5.4.3 and the induction hypothesis that the statement is true in state S_{k+1}. For all other data items x' such that $x' \neq x$ and $Page(x') \neq Page(x)$, the statement is true in state S_{k+1} by the induction hypothesis.

Case 2.1. $S_k.DirtyPages[Page(x)] = \emptyset$. In this case, the page that the recorded operation affected is not missing and from Lemma 5.4.3 it follows that the statement is true in state S_{k+1}.

Case 2.2. $S_k.DirtyPages[Page(x)] \neq \emptyset$ and $S_k.DirtyPages[Page(x)].RecLSN > S_k.RedoLSN$. The log record that the redo pass is currently processing does not require redo even though the page is dirty, as it is the effects of some later operation that caused the page to be dirty.

From the effects of the action, $S_{k+1}.RedoLSN = S_k.RedoLSN + 1$

$$< S_k.DirtyPages[Page(x)].RecLSN + 1$$

$$< S_{k+1}.DirtyPages[Page(x)].RecLSN + 1$$

$$\leq S_{k+1}.DirtyPages[Page(x)].RecLSN$$

Also, $S_{k+1}.DB[Page(x)].x = S_k.DB[Page(x)].x$, therefore $S_{k+1}.MaxLSN(P) = S_k.MaxLSN(P)$. Hence by the induction hypothesis, the statement is true in state S_{k+1}.

Case 2.3. $S_k.DirtyPages[Page(x)] \neq \emptyset$, $S_k.RedoLSN \geq S_k.DirtyPages[Page(x)].RecLSN$ and $S_k.DB[Page(x)].PageLSN < S_k.RedoLSN$. In this case, the effects of the operation recorded in the log record that the redo pass is currently processing are missing. The actions then *redo* the operation. In this case, $S_{k+1}.MaxLSN(Page(x)) = S_{k+1}.RedoLSN - 1 = S_k.RedoLSN$. From the effects of the action π_{k+1},

$$S_{k+1}.DB[Page(x)].x = S_k.Log[S_k.RedoLSN].Data.AfterImage.$$

Therefore

$$S_{k+1}.DB[Page(x)].x =$$

$$S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.MaxLSN(Page(x)), x)].Data.AfterImage.$$

Hence, the statement is true in state S_{k+1}.

Case 2.4. $S_k.DirtyPages[Page(x)] \neq \emptyset$, $S_k.RedoLSN \geq S_k.DirtyPages[Page(x)].RecLSN$ and $S_k.DB[Page(x)].PageLSN \geq S_k.RedoLSN$. The final case is where the dirty pages table indicates that the recorded operation (in the log record) that the redo pass is currently processing is potentially missing. However, when the page that the operation affected is fetched and the page's *PageLSN* is examined, the *redo* pass determines that the effects are not missing. In this case, $S_k.MaxLSN(Page(x)) = S_k.DB[Page(x)].PageLSN$. From the effects of the action π_{k+1},

$$S_{k+1}.RedoLSN = S_k.RedoLSN + 1,$$

$$S_{k+1}.DB[Page(x)] = S_k.DB[Page(x)] \quad \text{and}$$

$$S_{k+1}.DirtyPages[Page(x)].RecLSN = S_{k+1}.DB[Page(x)].PageLSN + 1.$$

Therefore $S_{k+1}.DB[Page(x)].PageLSN \geq S_{k+1}.RedoLSN - 1$. Hence $S_{k+1}.MaxLSN(Page(x)) = S_k.MaxLSN(Page(x))$. Therefore by the inductive hypothesis the statement is true in state S_{k+1}.

Case 3. $\pi_{k+1} \notin \{AnalysisTerm, Redo\}$. The action modifies the database if and only if it recorded the operation in the log. Thus it is elementary to show that the statement is true in state S_{k+1}. This completes the induction. □

D. Proof of Proposition 5.5.5

It is elementary to show that the statement is true in state S_0.

Assume that the statement is true in state S_k. To complete the induction, we must show that the statement is true in state S_{k+1}. The nontrivial cases are when a transaction terminates or normal operation is resumed; that is,

$$\pi_{k+1} \in \{AckCommit(T), AckRollbk(T), AckRestart\}.$$

Case 1. $\pi_{k+1} = AckCommit(T)$. Since all active transactions are rolled back after a crash, it follows that no crash has occurred since the transaction became active. In this case, the transaction T successfully commits. It follows from the definitions that the value the transaction wrote to the data item x is the last committed value for x. From Proposition 5.4.5 and the definitions,

$$\begin{aligned} &S_{k+1}.DB[Page(x)].x \\ &\quad = S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].Data.AfterImage \\ &\quad = S_{k+1}.Log[S_{k+1}.Lcw(1, S_{k+1}.LSN - 1, x)].Data.AfterImage. \end{aligned}$$

Therefore the statement is true in state S_{k+1}.

Case 2. $\pi_{k+1} = AckRollbk(T)$. A transaction's rollback terminates when all the updates by the transaction have been undone. Therefore from

Lemma 5.5.4,

$$S_{k+1}.DB[Page(x)].x =$$

$$S_{k+1}.Log[S_{k+1}.Lcw(1, S_{k+1}.LSN - 1, x)].Data.AfterImage.$$

Therefore the statement is true in state S_{k+1}.

Case 3. $\pi_{k+1} = AckRestart$. From the model, it is elementary to show that $\exists c$ such that $\pi_c = Crash$ and $\forall c'$, $c < c' < k$, $\pi_{c'} \neq Crash$.

Let x be any data item and $T = S_c.Log[S_c.Lwr(1, S_c.LSN - 1, x)].TransID$.

Case 3.1. $\exists e$ such that $1 \leq e < S_c.LSN$, $S_c.log[e].Type = End$, $S_c.log[e].TransID = T$. In this case, the transaction that last wrote to x terminated and its *end* log record was flushed to the stable log. From Proposition 5.4.5,

$$S_{k+1}.DB[Page(x)].x =$$

$$S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].Data.AfterImage.$$

Let S_i be the state immediately after the action that wrote the *end* log record. Since T was the last transaction that wrote to the data item x, it follows from the induction hypothesis on state S_i that

$$\begin{aligned}
&S_{k+1}.DB[Page(x)].x\\
&\quad = S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].Data.AfterImage\\
&\quad = S_i.Log[S_{k+1}.Lwr(1, S_i.LSN - 1, x)].Data.AfterImage\\
&\quad = S_i.Log[S_{k+1}.Lcw(1, S_i.LSN - 1, x)].Data.AfterImage\\
&\quad = S_{k+1}.Log[S_{k+1}.Lcw(1, S_{k+1}.LSN - 1, x)].Data.AfterImage.
\end{aligned}$$

Hence the statement is true in state S_{k+1}.

Case 3.2. $\nexists e$ such that $1 \leq e < S_c.LSN$, $S_c.log[e].Type = End$, $S_c.log[e].TransID = T$. In this case, the transaction was either active before the crash or it terminated, but its *end* log record never made it out to the stable log. There are two cases to consider.

Case 3.2.1. $S_k.UpdateLSN(T, x) \neq -1$. In this case, the transaction recorded the update to x in the stable log before the crash. The proof is similar to Case 2 because the undo pass will undo the update and restore the last committed value.

Case 3.2.2. $S_k.UpdateLSN(T, x) = -1$. Due to the crash, either the *update* log record was never written or was lost. The cache manager enforces write-ahead logging, therefore the effects of the update to x by the transaction was not flushed out to the stable database. From Proposition 5.4.5,

$$S_{k+1}.DB[Page(x)].x =$$

$$S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].Data.AfterImage.$$

The action π_i requested a write to x and, since the execution is strict, it follows that the transaction $S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].TransID$ terminated before state S_i. From the inductive hypothesis on state S_i,

$$\begin{aligned} S_{k+1}.DB[Page(x)].x \\ &= S_{k+1}.Log[S_{k+1}.Lwr(1, S_{k+1}.LSN - 1, x)].Data.AfterImage \\ &= S_i.Log[S_i.Lwr(1, S_i.LSN - 1, x)].Data.AfterImage \\ &= S_i.Log[S_i.Lcw(1, S_i.LSN - 1, x)].Data.AfterImage \\ &= S_{k+1}.Log[S_{k+1}.Lcw(1, S_{k+1}.LSN - 1, x)].Data.AfterImage. \end{aligned}$$

This completes the induction. □

E. Proof of Correctness (Theorem 5.6.1)

The definition requires us to show the existence of a set $CommitTrans(\alpha)$, which we choose to be

$$CommitTrans(\alpha) =$$

$$\left\{ T : \begin{matrix} \exists i, 1 \le i \ge n, \pi_i = LogFlush[l], \\ S_i.Log[l], Type = End, S_i.Log[l].Data = Commit \end{matrix} \right\}.$$

We require to show for all r, $1 \le x \le n$, that if $\pi_r = AckRead(T, x, v)$, then

$$v = \begin{cases} v' & \text{if } \exists j, j < r, \pi_j = ReqWrite(T, x, v') \\ LastUpdate(CommitTrans(\alpha), x, n) & \text{otherwise} \end{cases}$$

Case 1. $\exists j, v'$ such that $j < n$ and $\pi_j = WriteReq(T, x, v')$. This case is when the transaction T has written to the data item x. Crashes terminate all transactions, therefore $\forall i, j \le i \le n, \pi_i \ne Crash$. Transactions are well-formed, thus we know that between the states S_j and S_n, the transaction stored the value v' to the data item x. Because executions are strict and no crashes have occurred since S_j, the read would have read the value v'. Hence $v = v'$.

Case 2. $\nexists j, v'$ such that $j < r$ and $\pi_j = WriteReq(T, x, v')$. It is elementary to show by induction that $\forall k, 0 \leq k$

$$S_k.Log[S_k.Lcw(1, S_k.LSN - 1, x)].Data.AfterImage$$

$$= LastUpdate(\alpha, CommitTrans(\alpha), x, k).$$

Executions are strict, therefore,

$$S_r.Log[S_r.Lwr(1, S_r.LSN - 1, x)].TransID \notin S_r.ActiveSet.$$

That is, the last writer transaction that wrote to the data item x has terminated. From Proposition 5.5.5,

$$v = S_r.Log[S_r.Lcw(1, S_r.LSN - 1, x)].Data.AfterImage.$$

Hence, $v = LastUpdate(\alpha, CommitTrans(\alpha), x, r)$.
Therefore α is correct. $\square$

ACKNOWLEDGMENTS

We are grateful to the anonymous referees and Alan Fekete for their valuable comments and suggestions, which have improved both the presentation and contents of this article.

REFERENCES

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA.

FEKETE, A. 1993. Formal models of communication services: A case study. *IEEE Comput. 26,* 8 (Aug.), 37–47.

FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. 1990. Commutativity-based locking for nested transactions. *J. Comput. Syst. Sci. 41,* 1 (Aug.), 65–156.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann, San Mateo, CA.

HADZILACOS, V. 1988. A theory of reliability in database systems. *J. ACM 35,* 121–145.

HARDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15,* 287–317.

KUO, D. 1992a. Model and verification of a data manager based on ARIES. In *Proceedings of the 4th International Conference on Database Theory,* 231–245.

KUO, D. 1992b. Model and verification of a data manager based on ARIES. Tech. Rep. SSRC 22.3, Basser Dept. of Computer Science, Univ. of Sydney, Sydney, Australia.

KUO, D. AND FEKETE, A. 1992. Model and verification of a data manager. In *Proceedings of the 15th Australian Computer Science Conference.*

KUO, D. AND FEKETE, A. 1994. Verification of a recovery algorithm using WAL. Tech. Rep. 292 (Feb.), Dept. of Computer Science, Univ. of Queensland, St Lucia, Australia.

LYNCH, N., MERRITT, M., WEIHL, W., AND FEKETE, A. 1993. *Atomic Transactions.* Morgan-Kaufmann, San Mateo, CA.

LYNCH, N. AND TUTTLE, M. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computation,* 137–151. Expanded version available as Tech. Rep. MIT/LCS/TR-387, Laboratory for Computer Science, MIT, Cambridge, MA, April.

MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17,* 1 (March), 94–162.

SELINGER, P. 1987. Chicken and eggs: The interrelationship of systems and theory. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium of Principles of Database Systems.*

Received March 1994; revised February 1996; accepted February 1996