Specification and Verification of the Undo/Redo Algorithm for Database Recovery

Yuri Gurevich Charles Wallace

July 12, 1995

Abstract

The undo/redo database recovery algorithm of [Gray 78] is specified in terms of an evolving algebra ([Gur 95]). The specification is used to verify correctness and liveness properties of the algorithm.

1 Introduction

In this paper, we introduce **evolving algebras** ([Gur 95]) as a formal specification method for database recovery. We specify the **undo/redo** recovery algorithm ([Gray 78]) in terms of an evolving algebra. Using this specification, we then prove correctness and liveness properties. The treatment of this simple algorithm is intended to be a starting point from which investigation of more complicated recovery techniques can proceed.

The information managed by most large-scale database systems is accessed and updated concurrently by multiple users, so the system must be able to service users' requests promptly. At the same time, this information is often of a critical nature, so data loss due to errors must be minimized. Therefore, a database recovery mechanism must operate as efficiently as possible during normal processing and during error recovery, while guaranteeing persistence of data in the face of system errors.

Recovery algorithms of varying levels of sophistication have been proposed to meet these goals. Advanced recovery algorithms are quite complex, and many exploit subtle properties of the other components of the database system. The current literature on the subject tends to rely on informal descriptions of the algorithms, and there has not been much effort to prove their correctness. The area of database recovery is clearly in need of formal techniques of specification and verification, not only to ensure that the algorithms work correctly, but also to formalize ideas that have never been explained satisfactorily.

We feel that evolving algebras are well suited for specifying database recovery concepts. Evolving algebras can be used to implement an algorithm at any level of abstraction ([Gur 93]); this allows us to specify the undo/redo algorithm at a high level, without concerning ourselves with implementation-specific details. Our results therefore apply to any implementation of the algorithm. This generic view can be made as specific as desired through subsequent refinements of the evolving algebra.

Acknowledgements. We are grateful to Nandit Soparkar for introducing us to the area of database recovery. We would also like to thank Jim Huggins for his comments on earlier drafts of this paper.

2 Problem description

Our first task is to state our verification conditions: what exactly must the recovery algorithm do? We begin by introducing some basic terms related to database management. A **database** is a set of data items, each identified by a **location** and containing a **value**. Data items are kept in both **stable** and **volatile** storage. Stable storage is a failure-resistant medium, while volatile storage is a medium allowing faster access than stable storage but prone to failure. Typically, a database system stores data in memory and on disk; the

memory storage would be termed volatile and the disk storage stable. In this paper we focus on recovery from failure of the volatile storage component; the effect of such a failure is called a **system error**.

Since writing to volatile storage is relatively fast, it is used for temporary storage of data items. In most systems, when a data item is updated the new value is stored first in volatile storage and later written to stable storage. Volatile storage is usually too small to accommodate the entire database, so only part of the database resides in volatile storage at any given time. As the system processes accesses and updates, copies of data items are continually written to volatile storage, then erased after being copied to stable storage. The current value of a data item is the value in volatile storage if a copy of the item is there; otherwise, the current value is the value in stable storage.

The database is accessed concurrently by multiple **application programs**. A program called the **database management system** or **DBMS**, provides an interface between the database and the application programs accessing it. Each application program may read and write values to the database; the sequence of reads and writes issued by an application program is called a **transaction**. Once it has finished accessing the database, the application program explicitly **commits** or **aborts** the transaction. By committing, the program indicates that it has terminated normally, and that the changes it made to the database should be maintained. The program aborts its transaction if it terminates abnormally; this indicates that the changes it made to the database should be **rolled back** or **undone**, *i.e.*, any data item with a value written by the aborting transaction should be reassigned its previous value.

The effects of a transaction on the database are therefore "all or nothing": either all or none of the transaction's changes are maintained in the database. This property of transactions is called **atomicity**. Furthermore, the effects of a committed transaction are persistent; the values written by a committed transaction are maintained even after system errors. This property is called **durability**. A database system which guarantees atomicity and durability of all its transactions simplifies each application program's interface with the database. Because of atomicity, application programs need not consider the possibility of "partially committed" or "partially aborted" transactions. The data available to an application is never data written by an erroneous transaction; the DBMS, rather than the application, identifies and eliminates erroneous data. Moreover, durability frees applications with committed transactions from having to enforce the persistence of their effects; it becomes the task of the DBMS rather than the application programs to perform recovery after an error.

The task of ensuring atomicity and durability is performed by a component of the DBMS called the **recovery manager** or **RM**. The RM receives a sequence of read, write, abort and commit commands; as transactions execute concurrently, commands from different transactions may be interleaved. Another component of the DBMS called the **scheduler** controls the sequence or **schedule** of transactions issued to the RM. The RM controls the contents of volatile storage (also called the **cache**) and of stable storage. Items to be read are **fetched**, or copied from stable storage to the cache. Items to be written may be copied directly to stable storage or to volatile storage. A component of the DBMS called the **cache manager** or **CM** periodically **flushes** data items from the cache, removing them after copying them to stable storage.

When a system error occurs the DBMS, including the RM and CM, stops running, and the contents of volatile storage are lost. It is assumed that a system error is a **fail-stop** type of error: when the DBMS restarts, it can detect that the error occurred and can pass into a recovery mode. Only the values saved in stable storage survive a system error. This leads to two types of inconsistency that the RM must rectify. First, if updates issued by a committed transaction were cached but never flushed to stable storage before the error, the committed changes to these items will no longer be recorded in the database after the error. To ensure durability, the RM must reinstate or **redo** the committed updates to the database. Second, if data items written by an uncommitted transaction were cached and subsequently flushed to stable storage before the error, the values of uncommitted transactions will be in stable storage after the error. To allow such transactions to continue would risk violating atomicity: between the time of the system error and the restart of the DBMS, the uncommitted transactions may have issued further write commands, all of which will have been unsuccessful. To ensure that a transaction's changes are never only partially recorded, the RM must undo the uncommitted changes recorded in stable storage.

The goal of the recovery procedure can then be stated simply: to reestablish the last committed value

of each item in the database. To show **correctness**, we must consider the last committed value of a data item at the time of a system error; this must be the value in the database immediately after recovery. In addition, we must prove **liveness**: there must be some bound on how long it takes the system to return to normal processing after an error. Of course, there are pathological cases in which liveness is impossible: the system may experience an arbitrarily long sequence of errors, with very little or no time between each error. In our proof of liveness, we focus on cases with error-free intervals sufficiently long to allow recovery.

3 Algorithm Description

The undo/redo algorithm originated in [Gray 78]; we use the presentation in [BHG 87] as our guidelines. One of the advantages of this algorithm, besides its simplicity, is the fact that it imposes no restrictions on the CM. Newly written data items stored in the cache are flushed only when the CM sees fit to do so. This allows the RM to service read and write commands without waiting for the CM to write to stable storage. On the other hand, it introduces some problems for data persistence: uncommitted values may have been flushed to stable storage and committed values may still reside only in the cache at the time of an error. The task of the recovery procedure is twofold: in the event of a system error, it must undo all changes made by uncommitted transactions, and it must redo all changes made by committed transactions.

As the contents of volatile storage are lost after a system error, the information that the recovery procedure needs to perform these undoes and redoes must reside somewhere in stable storage. There must be a record of the write commands issued before the error, and there must be a record of the transactions that committed before the error. In the undo/redo algorithm, the RM maintains a **log** of write commands in stable storage during normal processing. Each log record consists of the ID of the transaction T performing the write, the location λ it writes to, the value v_{new} being written, and the value v_{old} in the database just before the write. This last item is called the **before image of λ with respect to T**: the value stored in the database at location λ before transaction T overwrote it.[1] In addition, the RM maintains a **commit list** in stable storage, consisting of the IDs of all committed transactions.

It is assumed that the schedule of transaction commands sent by the scheduler to the RM is **strict**. In a strict schedule, no read or write command to a location λ occurs until after all transactions that previously wrote to λ have either committed or aborted; this implies that no transaction ever reads an uncommitted value. To see why this is important, consider a situation in which a transaction T_1 writes a value at location λ, another transaction T_2 reads this value, and T_1 subsequently aborts. T_2 would then be forced to abort as well, as the value it read for λ would no longer be valid. This phenomenon is avoided if schedules are strict; individual transactions can be aborted in isolation, without affecting the termination status of other transactions.

Strict schedules have another attractive quality. When processing transaction T's abort and undoing the effect of its write to λ, the RM must replace the current value with the value that would have been at λ had T never executed. If schedules are not necessarily strict, this value may be difficult to determine. Consider a schedule in which T_1, T_2 and T_3 write to λ successively. If T_3 aborts and the recovery procedure undoes its write to λ, it must check the termination status of T_1 and T_2. If neither transaction aborted, the value to write is the before image of λ with respect to T_3: the value of T_2's write to λ. On the other hand, if T_2 aborted, the value to write is the before image of λ with respect to T_2: the value of T_1's write. If both T_1 and T_2 aborted, there is yet another value to be written: the before image of λ with respect to T_1. As the number of writes to λ increases, so does the number of transactions to check in case of an abort. However, if the schedule is strict, only one transaction must be checked: T_2, in our example. T_2 either committed or aborted before T_3's abort, so the value to write is either T_3's before image (if T_2 committed) or T_2's before image (if T_2 aborted). Just as every read in a strict schedule reads a committed value, so every write in

[1] In [BHG 87]'s abstract description of the algorithm, log records are not assumed to include before images; it is simply stated that any before image "can be found in the log." In the more specific description, before images are explicitly included in the log records, as "keeping such information in these records greatly facilitates the processing of RM-Abort and Restart [*i.e.*, abort and recovery processing]." We adopt this assumption in our specification.

a strict schedule writes over a committed value. Thus every before image is a committed value and so the proper value to write in the case of an undo.

The RM accepts a strict schedule of commands during normal processing. It processes a write by caching the new value and adding a log record including the new value and the before image. Processing a read command consists simply of a fetch of the requested data item. When a transaction commits, the transaction's ID is added to the commit list. When a transaction aborts, the RM scans back through the log, searching for entries with the transaction's ID. For each such record that it finds, it undoes the write by caching the before image.

The recovery procedure consists of a similar scan through the log. First the RM checks the location associated with the current log record to determine whether the data item has already been undone or redone. It it has not, the RM searches for the transaction ID of the log entry in the commit set. If it is there, the write corresponding to this log entry was committed, and so it is redone: the after image is cached. Otherwise, the write was not committed before the error, and so it is undone: the before image is cached. When the entire log has been scanned, the recovery procedure ends and normal processing continues.

4 Evolving Algebras

Before specifying the undo/redo algorithm formally, we provide a brief description of the evolving algebra concepts we will need. A **sequential evolving algebra** (hereafter, **ealgebra**) models a system in which a single agent changes the current state in discrete steps. The behavior of the system can be seen as a sequence of states, with each noninitial state determined by its predecessor in the sequence. To model such systems, a specification method must define what a state is and how a state is obtained from its predecessor. We explain the ealgebra notion of state first and then the notion of state transition rules.

An ealgebra has a **vocabulary**, which is a finite collection of function names, each of a fixed arity. A function may have an arity of zero; such a function is called **nullary**. There are certain function names that appear in every ealgebra's vocabulary: the nullary function names $true$, $false$ and $undef$, the equality sign, and the names of the usual Boolean operators.

A **state** S of an ealgebra E with vocabulary Υ consists of a nonempty set X, called the **superuniverse** of S, and an interpretation of each function name in Υ over X. The superuniverse is sorted into **universes**: for the domain $Dom(f)$ of any function f in Υ, there is a unary function U_f over X which returns $true$ for any member of $Dom(f)$ and $false$ for any other element of X. The name $undef$ is used to represent partial functions: for any tuple outside its domain, a partial function returns $undef$. The Boolean operators behave in the expected way over $\{true, false\}$ and return $undef$ for all other inputs.

Some functions have interpretations that do not change during any execution of the ealgebra; such functions are called **static**. The functions $true$, $false$ and $undef$, equality, and the Boolean operators are all static functions. On the other hand, the dynamic nature of a system is captured by functions whose interpretations do change over the course of an execution. Such functions are called **dynamic**. Of these, some functions change in a way determined by the state of the system; these functions are called **internal**. Others may change in ways beyond the system's control; these represent external forces which affect the system. Such functions are called **external**.

While external functions define uncontrollable actions affecting the system, **transition rules** define the system dynamics that are within the control of the system. An **update instruction**, the simplest type of transition rule, has the form $f(\bar{x}) := v$, where f is a dynamic function name of some arity n, $\bar{x}$ is an n-tuple of terms, and v is a term. Executing an update instruction changes the function at the given value; if $\bar{a}$ and b are of the values of $\bar{x}$ and v in a given state, then $f(\bar{a}) = b$ in the succeeding state.

A sequence of transition rules is itself a transition rule and is called a **rule block**. Executing a rule block is done by executing each of its transition rules simultaneously. If two different rules **conflict**, *i.e.*, attempt to change the same function at the same value, the ealgebra halts.

A **conditional instruction** is a transition rule of the form

if g_0 then
    R_0
elseif g_1 then
    R_1
    $\vdots$
elseif g_n then
    R_n
endif

where each **guard** g_i is a Boolean first-order term and each R_i is a transition rule.[2] Executing a conditional instruction results in the execution of a transition rule R_i, where i is the minimal value for which g_i evaluates to *true*. If no guard evaluates to *true*, the conditional instruction performs no execution of any R_i.

If Υ is the vocabulary of an ealgebra E, then the set Υ^- of all internal function names of E is E's **internal vocabulary**. If S is a state of E, then S^- is the corresponding **internal state**, *i.e.*, the state identical to S but with vocabulary Υ^-. A **run** of the ealgebra is a sequence of **stages**, each stage $\mathcal{S}$ consisting of a state of the ealgebra and its number $I(\mathcal{S})$ in the sequence. For each stage $\mathcal{S}$ after the initial stage, the internal state of $\mathcal{S}$ is obtained from the state of the previous stage by executing all enabled updates of E simultaneously.

For any term t and any stage $\mathcal{S}$, we take $[t]_{\mathcal{S}}$ to be the result of evaluating t at stage $\mathcal{S}$. In the case of a **static term** (*i.e.*, a term in which no dynamic function symbol appears), we simply use t to denote the result of evaluating t at an arbitrary stage in the run. For brevity, we extend the use of comparison operators to stages: for example, we take $\mathcal{S} < \mathcal{T}$ to mean $I(\mathcal{S}) < I(\mathcal{T})$. We use the notation $\mathcal{S} + n$, where n is an integer, to refer to the stage $\mathcal{T}$ for which $I(\mathcal{T}) = I(\mathcal{S}) + n$. We use interval notation to denote subsequences of a run; for example, $[\mathcal{S}, \mathcal{T})$ refers to the subsequence of the run containing all stages $\geq \mathcal{S}$ and $< \mathcal{T}$. Finally, we say that a function is **unchanged** in an interval if it evaluates to the same value at all stages in that interval.

5 Specification

Having described the undo/redo algorithm informally, we now specify it in terms of an ealgebra. We begin by defining the universes and functions of the vocabulary. In this ealgebra, $\emptyset$ and $\boldsymbol{Nil}$ denote the empty set and sequence, respectively. We use the operator $\in$ to represent membership in a set or sequence. For element x and set S, $\boldsymbol{S + x}$ is the set obtained by adding x to S, *i.e.*, $S \cup \{x\}$.

5.1 Vocabulary

First, we define the universes and static functions. We define a universe $\boldsymbol{TransIDs}$ of transactions, a universe of $\boldsymbol{Locations}$ of data items in the database, and a universe of $\boldsymbol{Values}$ that may be stored at these locations. A command issued to the recovery manager is of one of four possible types; we introduce the universe $\boldsymbol{Actions = \{Read, Write, Abort, Commit\}}$ to represent these. A command always includes a transaction ID and an action type; a read or write command includes a location, and a write command also includes a value. Therefore the universe $\boldsymbol{Commands}$ consists of quadruples: $\boldsymbol{TransIDs \times Locations \times Values \times Actions}$. We add projection functions mapping a given command to its associated components: $\boldsymbol{Trans : Commands \rightarrow TransIDs}$, $\boldsymbol{Loc : Commands \rightarrow Locations}$, $\boldsymbol{Val : Commands \rightarrow Values}$ and $\boldsymbol{Act : Commands \rightarrow Actions}$.

A log entry specifies a transaction performing a write, the location to which the transaction is writing, the new value to be written, and the old value to be overwritten. We define a universe $\boldsymbol{Entries}$ of quadruples: $\boldsymbol{TransIDs \times Locations \times Values \times Values}$. We add the appropriate projection functions: $\boldsymbol{Trans : Entries \rightarrow TransIDs}$, $\boldsymbol{Loc : Entries \rightarrow Locations}$, $\boldsymbol{NewVal : Entries \rightarrow Values}$

[2] In a nested conditional instruction, a sequence of endif's may be abbreviated as ENDIF.

and $\mathbf{OldVal : Entries \rightarrow Values}$. We represent a log configuration as an element of the universe $\mathbf{LogConfigs}$, which consists of all finite sequences of elements over *Entries*.

Each new log entry is added to the end of the log. $\mathbf{WriteLog : TransIDs \times Locations \times Values \times Values \times LogConfigs \rightarrow LogConfigs}$ takes a transaction, location, new value and old value and returns the result of adding the resulting log entry to the end of the log. $\mathbf{LogEnd : LogConfigs \rightarrow Entries}$ takes a log configuration and returns the entry at its end; we assume that $LogEnd(Nil)$ evaluates to *undef*. $\mathbf{PrevEntry : Entries \times LogConfigs \rightarrow Entries}$ returns the entry immediately preceding the given entry in the log. If the given entry is the first entry in the log, *PrevEntry* returns *undef*. $\mathbf{EntryPos : Entries \times LogConfigs \rightarrow PosInt}$ returns the number of entries between the given entry and the end of the log. $\mathbf{LastEntry : Locations \times LogConfigs \rightarrow Entries}$ returns the entry nearest the end of the log with the given location.[3]

We represent a cache configuration as an element of the universe $\mathbf{CacheConfigs}$. Each entry in the cache is a data item consisting of a location and a value. The cache contains at most one copy of a given data item: *i.e.*, no two cache entries have the same location. Therefore, *CacheConfigs* consists of all finite subsets of $(Locations \times Values)$ for which no subset contains two elements with the same location.

$\mathbf{InCache : Locations \times CacheConfigs \rightarrow \{true, false\}}$ returns *true* if the cache contains an entry with the given location. $\mathbf{ReadCache : Locations \times CacheConfigs \rightarrow Values}$ returns the result of reading a data item of the given location from the given cache state. $\mathbf{WriteCache : Locations \times Values \times CacheConfigs \rightarrow CacheConfigs}$ returns the cache state achieved after adding a data item of the given location and value to the cache. A data item is overwritten if it is already in the cache: if a cache configuration C contains a data item (λ, v), then $WriteCache(\lambda, v', C)$ returns the result of replacing (λ, v) in C with (λ, v'). $\mathbf{Flush : Locations \times CacheConfigs \rightarrow CacheConfigs}$ returns the cache state resulting from flushing a data item. If the given location is *undef* or there is no data item with the given location in the cache, the function returns the cache configuration unchanged.

The recovery manager maintains in stable storage a list of transactions that have committed. If a transaction is in this list, its writes will be redone during recovery; otherwise, they will be rolled back. We model the commit list as an element of the universe $\mathbf{TransSets}$, consisting of all finite subsets of *TransIDs*. In addition, the recovery procedure keeps track of the set of locations it has either undone or redone; if a particular location has already been undone or redone, the recovery procedure ignores all subsequent log entries with that location. We introduce the universe $\mathbf{LocSets}$, consisting of all finite sets of elements over *Locations*.

We now define the dynamic functions which determine the state of the RM; each function is internal unless specified as external. The external function $\mathbf{Error : \{true, false\}}$ returns *true* if a system-level error has just occurred. $\mathbf{InRecovery : \{true, false\}}$ is set to *true* when a system-level error is detected and recovery is initiated. $\mathbf{Cmd : Commands}$ represents the command currently being processed. An external function $\mathbf{NextCmd : Commands}$ represents the command scheduled immediately after the current one. $\mathbf{Stable : Locations \rightarrow Values}$ represents the current state of all data items as they are recorded in stable storage. $\mathbf{Log : LogConfigs}$ represents the current log contents, and $\mathbf{Cache : CacheConfigs}$ represents the current contents of the cache. An external function $\mathbf{Victim : Locations}$ represents the location of the data item selected by the CM to be flushed from the cache. $\mathbf{CurrEntry : Entries}$ returns the log entry currently being considered by the recovery procedure. $\mathbf{CommitSet : TransSets}$ represents the current set of committed transactions. Finally, $\mathbf{Corrected : LocSets}$ represents the set of undone and redone locations.

5.2 Rules for normal processing

Our first rules are shown in Fig. 1. The rule $\mathbf{WriteToStable}$ represents the write to stable storage caused by a cache flush. There may be no victim to flush at a given state, in which case the value of *Victim*

[3] The functions *PrevEntry* and *EntryPos* return *undef* if the given entry does not appear or appears more than once in the given log configuration. The function *LastEntry* returns *undef* if no entry with the given location appears in the given log configuration.

rule $WriteToStable$:
if not $Error$ then
    if $Victim \neq undef$ and $InCache(Victim, Cache)$ then
        $Stable(Victim) := ReadCache(Victim, Cache)$
ENDIF

rule $NoCommand$:
if $Act(Cmd) = undef$ and not ($Error$ or $InRecovery$) then
    $Cache := Flush(Victim, Cache)$
    $Cmd := NextCmd$
endif

Figure 1: Rules $WriteToStable$ and $NoCommand$

is $undef$ and therefore $WriteToStable$ makes no update. If no command is issued to the RM, the rule ***NoCommand*** simply updates the current command. Note that although no RM processing occurs in this case, the CM may flush a data item.

The rules for processing read and write commands appear in Fig. 2. The rule ***DoRead*** simply fetches the requested data item into the cache, if it is not already there. The rule ***DoWrite*** fetches the before image into the cache, if it is not already there, and then writes a new value to the cache, adding a log entry recording the write.

The rules for processing commits and aborts are shown in Fig. 3. The rule ***DoCommit*** simply adds the given transaction to the commit list. The rule ***DoAbort*** scans the log, undoing each of the aborting transaction's writes by writing each log entry's old value to cache, until the list is empty.

5.3 Rules for recovery processing

We define rules to represent the occurrence of a system-level error and the ensuing recovery procedure. The rule ***Restart***, shown in Fig. 4, models the system's initial actions after a system-level error. The cache is erased, the set of undone and redone locations is set to empty, the log index is set to the end of the log, and the recovery flag is set to *true*. The rules ***ScanLog*** and ***DoCorrect***, shown in Fig. 5, model the process of performing the necessary undoes and redoes after a system-level error. The log index is decremented by *ScanLog* at each step in the recovery, so that the log is examined from end to beginning. Each uncorrected entry encountered causes *DoCorrect* to fire, writing either the before image or after image of the update to the cache, depending on whether the update is committed.

6 Verification

Our verification proceeds as follows. To simplify the proofs, we introduce some terms based on the specification in the previous section. Our first claims are easily verifiable and presented without proof. We then make a couple of claims about the structure of the log; following this, we present claims about the processing of aborts, then some general claims about normal (nonrecovery) processing, and finally a couple of claims about recovery processing. Using these, we present three lemmata which provide the basis for the correctness proof. Finally, our two theorems assert the desired correctness and liveness properties. Throughout this section, we use the following variables: let $\mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{S}', \mathcal{T}$ be stages, and let $\lambda, \lambda' \in Locations$, $t \in TransIDs$, $E, E' \in Entries$, and $v, v_{old}, v_{new} \in Values$.

rule $DoRead$:
if $Act(Cmd) = Read$ and not ($Error$ or $InRecovery$) then
    if not $InCache(Loc(Cmd), Cache)$ then
        $Cache := WriteCache(Loc(Cmd), Val(Cmd), Flush(Victim, Cache))$
    else
        $Cache := Flush(Victim, Cache)$
    endif
    $Cmd := NextCmd$
endif

rule $DoWrite$:
if $Act(Cmd) = Write$ and not ($Error$ or $InRecovery$) then
    if not $InCache(Loc(Cmd), Cache)$ then
        $Cache := WriteCache(Loc(Cmd), Stable(Loc(Cmd)), Flush(Victim, Cache))$
    else
        $Log := WriteLog(Trans(Cmd), Loc(Cmd), Val(Cmd), ReadCache(Loc(Cmd), Cache), Log)$
        $Cache := WriteCache(Loc(Cmd), Val(Cmd), Flush(Victim, Cache))$
        $Cmd := NextCmd$
ENDIF

Figure 2: Rules $DoRead$ and $DoWrite$

6.1 Definitions

Definition

- A stage $\mathcal{S}$ is an **error stage** if $[Error]_{\mathcal{S}} = true$.
- A nonerror stage $\mathcal{S}$ is a **recovery stage** if $[InRecovery]_{\mathcal{S}} = true$.
- A nonerror, nonrecovery stage $\mathcal{S}$ is **normal**.

Definition

- A normal stage $\mathcal{S}$ is a **commit stage** if $[Act(Cmd)]_{\mathcal{S}} = Commit$. We define **abort stage** similarly.
- A commit or abort stage is a **terminating stage**.
- A normal stage $\mathcal{S}$ is a **read stage** if $[Act(Cmd)]_{\mathcal{S}} = Read$, or if $[Act(Cmd)]_{\mathcal{S}} = Write$ but $[InCache(Loc(Cmd), Cache)]_{\mathcal{S}} = false$. (Note: in either case, the data item specified by $Loc(Cmd)$ is fetched from stable storage into the cache.)
- A normal stage $\mathcal{S}$ is a **write stage** if $[Act(Cmd)]_{\mathcal{S}} = Write$ and $[InCache(Loc(Cmd), Cache)]_{\mathcal{S}} = true$.

Definition

- A ***t*-stage** is a normal stage for which $[Trans(Cmd)]_{\mathcal{S}} = t$, or a recovery stage for which $[Trans(CurrEntry)]_{\mathcal{S}} = t$.
- A **λ-stage** is a read or write stage for which $[Loc(Cmd)]_{\mathcal{S}} = \lambda$, or an abort or recovery stage for which $[Loc(CurrEntry)]_{\mathcal{S}} = \lambda$.

```
rule DoCommit:
if Act(Cmd) = Commit and not (Error or InRecovery) then
    CommitSet := CommitSet + Trans(Cmd)
    Cache := Flush(Victim, Cache)
    Cmd := NextCmd
endif

rule DoAbort:
if Act(Cmd) = Abort and not (Error or InRecovery) then
    if Log = Nil then
        Cache := Flush(Victim, Cache)
        Cmd := NextCmd
    elseif CurrEntry = undef then
        CurrEntry := LogEnd(Log)
        Cache := Flush(Victim, Cache)
    else
        if Trans(CurrEntry) = Trans(Cmd) then
            Cache := WriteCache(Loc(CurrEntry), OldVal(CurrEntry), Flush(Victim, Cache))
        else
            Cache := Flush(Victim, Cache)
        endif
        CurrEntry := PrevEntry(CurrEntry, Log)
        if PrevEntry(CurrEntry, Log) = undef then
            Cmd := NextCmd
ENDIF
```

Figure 3: Rules *DoCommit* and *DoAbort*

```
rule Restart:
if Error then
    Cache := ∅
    Corrected := ∅
    CurrEntry := LogEnd(Log)
    InRecovery := true
endif
```

Figure 4: Rule *Restart*

rule $ScanLog$:
if $InRecovery$ and not $Error$ then
    if $CurrEntry \neq undef$ then
        $CurrEntry := PrevEntry(CurrEntry, Log)$
    else
        $InRecovery := false$
        $Cmd := NextCmd$
ENDIF

rule $DoCorrect$:
if $InRecovery$ and not $Error$ and $CurrEntry \neq undef$ then
    if $Loc(CurrEntry) \notin Corrected$ then
        $Corrected := Corrected + Loc(CurrEntry)$
        if $Trans(CurrEntry) \in CommitSet$ then
            $Cache := WriteCache(Loc(CurrEntry), NewVal(CurrEntry), Flush(Victim, Cache))$
        else
            $Cache := WriteCache(Loc(CurrEntry), OldVal(CurrEntry), Flush(Victim, Cache))$
ENDIF

Figure 5: Rules $ScanLog$ and $DoCorrect$

Definition

- A recovery stage $\mathcal{S}$ is a **correcting recovery stage** if $[Trans(CurrEntry)]_{\mathcal{S}} = [Trans(Cmd)]_{\mathcal{S}}$ and $[Loc(CurrEntry)]_{\mathcal{S}} \notin [Corrected]_{\mathcal{S}}$. (Note: if these conditions are met, the recovery procedure "corrects" (*i.e.*, undoes or redoes) the write recorded in the log entry $[CurrEntry]_{\mathcal{S}}$.)
- An abort stage $\mathcal{S}$ is an **abort-undo stage** if $[Trans(Cmd)]_{\mathcal{S}} = [Trans(CurrEntry)]_{\mathcal{S}}$. (Note: if this condition is met, the abort procedure undoes the write recorded in the log entry $[CurrEntry]_{\mathcal{S}}$.)

Definition

- $\boldsymbol{Norm(\mathcal{S})}$ is the first normal stage following $\mathcal{S}$, *i.e.*, the minimal normal stage $\geq \mathcal{S}$.[4]
  $\boldsymbol{Err(\mathcal{S})}$ is the last error stage before $\mathcal{S}$, *i.e.*, the maximal error stage $< \mathcal{S}$.
- A nonempty interval $\big(\mathcal{S}, Norm(\mathcal{S})\big)$ is a **recovery phase**.
- We say that $\boldsymbol{t}$ **aborted before** $\boldsymbol{\mathcal{S}}$ if there is a t-abort stage $< \mathcal{S}$.
  $\boldsymbol{Abt(t)}$ is the initial stage of t's abort, *i.e.*, the minimal t-abort stage.

Definition

- $\boldsymbol{[LW(\lambda)]_{\mathcal{S}}}$ is the stage of the last write to λ before $\mathcal{S}$, *i.e.*, the maximal λ-write stage $\leq \mathcal{S}$.[5]
  $\boldsymbol{[LT(\lambda)]_{\mathcal{S}}}$ is the transaction issuing this write, *i.e.*, $[Trans(Cmd)]_{[LW(\lambda)]_{\mathcal{S}}}$.

[4] Note that the functions $Norm$, Err and Abt may be undefined for certain values.

[5] $[LW(\lambda)]_S$ may be undefined for a given λ, in which case $[LT(\lambda)]_S$ is also undefined. Likewise, $[LCW(\lambda)]_S$ may be undefined, in which case $[LCT(\lambda)]_S$ and $[LCV(\lambda)]_S$ are also undefined.

- $[\boldsymbol{LCW(\lambda)}]_{\mathcal{S}}$ is the stage of the last committed write to λ before $\mathcal{S}$, *i.e.*, the maximal λ-write stage $\mathcal{R} < \mathcal{S}$ for which there is a commit stage $\mathcal{S}'$ where $[Trans(Cmd)]_{\mathcal{S}'} = [Trans(Cmd)]_{\mathcal{R}}$.

  $[\boldsymbol{LCT(\lambda)}]_{\mathcal{S}}$ is the transaction issuing this write, *i.e.*, $[Trans(Cmd)]_{[LCW(\lambda)]_{\mathcal{S}}}$.

  $[\boldsymbol{LCV(\lambda)}]_{\mathcal{S}}$ is the value written by this write, *i.e.*, $[Val(Cmd)]_{[LCW(\lambda)]_{\mathcal{S}}}$.

Definition

- If $[Cache]_{\mathcal{S}+1} = [WriteCache(\lambda, v, Flush(Victim, Cache))]_{\mathcal{S}}$, we say that ***v* is written to cache in location $\boldsymbol{\lambda}$** at $\mathcal{S}$.
- If $[Log]_{\mathcal{S}+1} = [WriteLog(t, \lambda, v_{new}, v_{old}, Log)]_{\mathcal{S}}$, we say that
  - ***t*'s write to $\boldsymbol{\lambda}$ is logged** at $\mathcal{S}$;
  - **before image $\boldsymbol{v_{old}}$** and **after image $\boldsymbol{v_{new}}$ are logged** at $\mathcal{S}$; and
  - **entry *E* is logged** at $\mathcal{S}$, where E is the entry with $Trans(E) = t$, $Loc(E) = \lambda$, $NewVal(E) = v_{new}$, and $OldVal(E) = v_{old}$.

Definition

- $[\boldsymbol{DB(\lambda)}]_{\mathcal{S}}$ is the value stored at location λ in the database at stage $\mathcal{S}$, *i.e.*,

$$\begin{cases} [ReadCache(\lambda, Cache)]_{\mathcal{S}} & \text{if } [InCache(\lambda, Cache)]_{\mathcal{S}} = true; \\ [Stable(\lambda)]_{\mathcal{S}} & \text{otherwise.} \end{cases}$$

- $[\boldsymbol{BeforeImage(\lambda, t)}]_{\mathcal{S}}$ is the before image of λ with respect to t, as recorded in the log, *i.e.*, $OldVal(E)$, where E is the entry in $[Log]_{\mathcal{S}}$ such that $Loc(E) = \lambda$ and $Trans(E) = t$.[6]

Definition

- A run is **strict** if any transaction writing to a location during the run either aborts or commits before another read or write to that location occurs, *i.e.*, if for any (t, λ)-write stage $\mathcal{S}$ and any λ-write or λ-read stage $\mathcal{U} > \mathcal{S}$, there is a t-terminating stage $\in (\mathcal{S}, \mathcal{U})$.
- A run is **regular** if no transaction issues commands after aborting or committing, *i.e.*, if for every t-terminating stage $\mathcal{S}$ there is no t-stage $\mathcal{T} > \mathcal{S}$, and no transaction issuing commands before an error does so after the error, *i.e.*, for every t-stage $\mathcal{R}$ and error stage $\mathcal{S} > \mathcal{R}$, there is no t-stage $> \mathcal{S}$.

6.2 Preliminary claims

We henceforth restrict attention to strict, regular runs. Claims 1-6 are easily verified, and we present them without proof.

Claim 1 If $\mathcal{S}$ is a nonerror stage, then

$$[DB(\lambda)]_{\mathcal{S}+1} = \begin{cases} v & \text{if a value } v \text{ is written to cache in location } \lambda; \\ [DB(\lambda)]_{\mathcal{S}} & \text{otherwise.} \end{cases}$$

[6] $[BeforeImage(\lambda, t)]_{\mathcal{S}}$ may be undefined for certain λ and t.

Claim 2 If at stage $\mathcal{S}$, t's write to λ' is logged with a before image v, then

$$[BeforeImage(\lambda, t)]_{\mathcal{S}+1} = \begin{cases} v & \text{if } \lambda = \lambda'; \\ [BeforeImage(\lambda, t)]_{\mathcal{S}} & \text{otherwise.} \end{cases}$$

Claim 3 If a write to λ' is logged in an entry E at stage $\mathcal{S}$, then

$$[LastEntry(\lambda, Log)]_{\mathcal{S}+1} = \begin{cases} E & \text{if } \lambda = \lambda'; \\ [LastEntry(\lambda, Log)]_{\mathcal{S}} & \text{otherwise.} \end{cases}$$

Claim 4 $LW(\lambda)$ is unchanged in $[\mathcal{S}, \mathcal{T}]$ if and only if there is no λ-write stage in $(\mathcal{S}, \mathcal{T}]$.

Claim 5 Log is unchanged in $[\mathcal{S}, \mathcal{T}]$ if there is no write stage in $[\mathcal{S}, \mathcal{T})$.

Claim 6 Let E be an entry for which $Trans(E) = t$ and $Loc(E) = \lambda$. Then $E \in [Log]_{\mathcal{S}}$ if and only if there is a (t, λ)-write stage $\mathcal{R} < \mathcal{S}$.

Claim 7 $[LT(\lambda)]_{\mathcal{S}} \in [CommitSet]_{\mathcal{S}+1}$ iff $[LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}$.

6.3 Claims involving log management

Claim 8 Let E and E' be entries for which $Trans(E) = Trans(E') = t$ and $Loc(E) = Loc(E') = \lambda$. If $E \in [Log]_{\mathcal{S}}$ and $E' \in [Log]_{\mathcal{S}}$, then $[EntryPos(E, Log)]_{\mathcal{S}} = [EntryPos(E', Log)]_{\mathcal{S}}$.

Proof. Assume to the contrary that $E \in [Log]_{\mathcal{S}}$ and $E' \in [Log]_{\mathcal{S}}$ but $[EntryPos(E, Log)]_{\mathcal{S}} \neq [EntryPos(E', Log)]_{\mathcal{S}}$. By Claim 6, there are (t, λ)-write stages $\mathcal{R} < \mathcal{S}$ and $\mathcal{R}' < \mathcal{S}$. As only one entry is logged at a single write stage, $\mathcal{R} \neq \mathcal{R}'$; without loss of generality, assume $\mathcal{R} < \mathcal{R}'$. If there is a t-terminating stage between $\mathcal{R}$ and $\mathcal{R}'$, the following (t, λ)-write stage $\mathcal{R}'$ violates regularity. If there is no such stage, strictness is violated. □.

Claim 9 $[Trans(LastEntry(\lambda, Log))]_{\mathcal{S}+1} = [LT(\lambda)]_{\mathcal{S}}$.

Proof. By induction on stages in $[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}]$.

Base step: $\mathcal{S} = [LW(\lambda)]_{\mathcal{S}}$. Since $\mathcal{S}$ is a λ-write stage, rule $DoWrite$ fires, logging the write by $[LT(\lambda)]_{\mathcal{S}}$ to λ. By Claim 3, $[Trans(LastEntry(\lambda, Log))]_{\mathcal{S}+1} = [LT(\lambda)]_{\mathcal{S}}$.

Inductive step: $\mathcal{S} > [LW(\lambda)]_{\mathcal{S}}$, so $[LW(\lambda)]_{\mathcal{S}} = [LW(\lambda)]_{\mathcal{S}-1}$ and therefore $[LT(\lambda)]_{\mathcal{S}-1} = [LT(\lambda)]_{\mathcal{S}}$. Assume that $[Trans(LastEntry(\lambda, Log))]_{\mathcal{S}} = [LT(\lambda)]_{\mathcal{S}-1}$. If $\mathcal{S}$ is a non-write stage, then by Claim 5, Log is unchanged in $[\mathcal{S}, \mathcal{S}+1]$. If $\mathcal{S}$ is a write stage, then it cannot be a λ-write stage, since $[LW(\lambda)]_{\mathcal{S}} < \mathcal{S}$. Rule $DoWrite$ fires, logging a write to a location $\neq \lambda$. By Claim 3, $LastEntry(\lambda, Log)$ is unchanged in $[\mathcal{S}, \mathcal{S}+1]$. In either case, by the inductive hypothesis, $[Trans(LastEntry(\lambda, Log))]_{\mathcal{S}+1} = [LT(\lambda)]_{\mathcal{S}}$. □.

6.4 Claims involving abort processing

Claim 10 If $\mathcal{S}$ is a (t, λ)-abort-undo stage, then $[CurrEntry]_{\mathcal{S}} = [LastEntry(\lambda, Log)]_{\mathcal{S}}$.

Proof. Assume to the contrary that $\mathcal{S}$ is a (t, λ)-abort-undo stage but $[CurrEntry]_{\mathcal{S}} \neq [LastEntry(\lambda, Log)]_{\mathcal{S}}$. Since $[Loc(LastEntry(\lambda, Log))]_{\mathcal{S}} = [Loc(CurrEntry)]_{\mathcal{S}} = \lambda$, by Claim 8 it must be that $t = [Trans(CurrEntry)]_{\mathcal{S}} \neq [Trans(LastEntry(\lambda, Log))]_{\mathcal{S}}$. By Claim 9, $t \neq [LT(\lambda)]_{\mathcal{S}}$. By Claim 6, there is a (t, λ)-write stage $\mathcal{Q} < \mathcal{S}$. Since $t \neq [LT(\lambda)]_{\mathcal{S}}$, $\mathcal{Q} < [LW(\lambda)]_{\mathcal{S}}$. If there is a t-commit stage $\mathcal{R}$, the presence of the t-abort stage $\mathcal{S}$ violates regularity. If there is no such $\mathcal{R}$, then there is no t-terminating stage between the λ-write stages $\mathcal{Q}$ and $[LW(\lambda)]_{\mathcal{S}}$, which violates strictness. □.

Claim 11 If $\mathcal{S}$ is a (t, λ)-abort-undo stage, then $t = [LT(\lambda)]_{\mathcal{S}}$.

Proof. Immediate from Claim 9 and Claim 10. □.

Claim 12 If $\mathcal{S}$ is a (t, λ)-abort-undo stage, then $[LW(\lambda)]_{\mathcal{S}} \neq [LCW(\lambda)]_{\mathcal{S}}$.

Proof. Assume to the contrary that $\mathcal{S}$ is a (t, λ)-abort-undo stage but $[LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}$. By Claim 11, $t = [LT(\lambda)]_{\mathcal{S}} = [LCT(\lambda)]_{\mathcal{S}}$. Then there is a t-commit stage before the t-abort stage $\mathcal{S}$, which violates regularity. □.

Claim 13 Let $\mathcal{S}$ be $Abt(t)$ for some t, let $E \in [Log]_{\mathcal{S}}$, and let $p = [EntryPos(E, Log)]_{\mathcal{S}}$. If $(\mathcal{S}, \mathcal{S} + p]$ is normal, then $[CurrEntry]_{\mathcal{S}+(p+1)} = E$.

Proof. By induction on p.

Base step: $p = 0$. Then $E = [LogEnd(Log)]_{\mathcal{S}}$. At $\mathcal{S}$, *DoAbort* fires, updating $CurrEntry$ so that $[CurrEntry]_{\mathcal{S}+1} = [LogEnd(Log)]_{\mathcal{S}} = E$.

Inductive step: $p > 0$. Assume $[CurrEntry]_{\mathcal{S}+p} = E'$, where $[EntryPos(E', Log)]_{\mathcal{S}} = p - 1$. Then $[PrevEntry(E', Log)]_{\mathcal{S}} = E$. At $\mathcal{S} + p$, *DoAbort* fires, updating $CurrEntry$ so that $[CurrEntry]_{\mathcal{S}+(p+1)} = [PrevEntry(E', Log)]_{\mathcal{S}} = E$. □.

Claim 14 Let $\mathcal{Q}$ be a (t, λ)-write stage $< \mathcal{S}$, let $\mathcal{S}$ be a nonabort stage, and let $[\mathcal{Q}, \mathcal{S}]$ be normal, If t aborts before $\mathcal{S}$, then there is a (t, λ)-abort-undo stage $\mathcal{R} \in (\mathcal{Q}, \mathcal{S})$.

Proof. Assume t aborts before $\mathcal{S}$. At $\mathcal{Q}$, t's write to λ is logged in entry E. Since the run is regular, the t-abort stage $Abt(t)$ is after the write stage $\mathcal{Q}$. Thus at $Abt(t) \in (\mathcal{Q}, \mathcal{S})$, $E \in [Log]_{Abt(t)}$, where E is an entry with $Trans(E) = t$ and $Loc(E) = \lambda$. Let $p = [EntryPos(E, Log)]_{Abt(t)}$, and let $\mathcal{R} = Abt(t) + (p + 1)$. By Claim 13, $[CurrEntry]_{\mathcal{R}} = E$, so $\mathcal{R}$ is a (t, λ)-abort-undo stage. □.

6.5 Claims involving normal processing

Claim 15 If $\mathcal{S}$ is a read or commit stage, then $DB(\lambda)$ is unchanged in $[\mathcal{S}, \mathcal{S} + 1]$.

Proof.

- If $\mathcal{S}$ is a λ'-read stage and $[InCache(\lambda', Cache)]_{\mathcal{S}} = true$, then *DoRead* fires but does not write to cache, so by Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{S}, \mathcal{S} + 1]$.

- If $\mathcal{S}$ is a λ'-read stage and $[InCache(\lambda', Cache)]_{\mathcal{S}} = false$, then if $\lambda' = \lambda$, $[DB(\lambda)]_{\mathcal{S}} = [Stable(\lambda)]_{\mathcal{S}}$, so *DoRead* writes $[DB(\lambda)]_{\mathcal{S}}$ to cache in location λ; by Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{S}, \mathcal{S}+1]$. If $\lambda' \neq \lambda$, then *DoRead* writes to cache in a location $\neq \lambda$; by Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{S}, \mathcal{S}+1]$.
- If $\mathcal{S}$ is a commit stage, then rule *DoCommit* fires but does not write to cache, so by Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{S}, \mathcal{S}+1]$. □.

Claim 16 Let $\mathcal{S}$ be a stage for which $[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}]$ is normal. If $\mathcal{S}+1$ is a λ-write stage and $[LW(\lambda)]_{\mathcal{S}} \neq [LCW(\lambda)]_{\mathcal{S}}$, then $[LT(\lambda)]_{\mathcal{S}}$ aborts before $\mathcal{S}$.

Proof. Assume to the contrary that $\mathcal{S}+1$ is a λ-write stage and $[LW(\lambda)]_{\mathcal{S}} \neq [LCW(\lambda)]_{\mathcal{S}}$, but $[LT(\lambda)]_{\mathcal{S}}$ does not abort before $\mathcal{S}$. Then there is an $[LCT(\lambda)]_{\mathcal{S}}$-commit stage $\mathcal{R} < \mathcal{S}$, but there is no $[LT(\lambda)]_{\mathcal{S}}$-terminating stage $\mathcal{R}' < \mathcal{S}$. If $\mathcal{R} < [LW(\lambda)]_{\mathcal{S}}$, then there is no $[LT(\lambda)]_{\mathcal{S}}$-terminating stage $\mathcal{R}'$ between the λ-write stages $[LW(\lambda)]_{\mathcal{S}}$ and $\mathcal{S}+1$. If $[LW(\lambda)]_{\mathcal{S}} < \mathcal{R}$, then there is no $[LCT(\lambda)]_{\mathcal{S}}$-terminating stage $\mathcal{R}'$ between the λ-write stages $[LCW(\lambda)]_{\mathcal{S}}$ and $[LW(\lambda)]_{\mathcal{S}}$. In either case, strictness is violated. □.

Claim 17 Let $\mathcal{S}$ be a nonabort stage for which $[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}]$ is normal, and let $t = [LT(\lambda)]_{\mathcal{S}}$. If t aborts before $\mathcal{S}$, then $[DB(\lambda)]_{\mathcal{S}} = [BeforeImage(\lambda, t)]_{\mathcal{S}}$.

Proof. Assume that t aborts before $\mathcal{S}$. By Claim 14, there is a (t, λ)-abort-undo stage $\mathcal{R} \in ([LW(\lambda)]_{\mathcal{S}}, \mathcal{S})$. Thus $[OldVal(CurrEntry)]_{\mathcal{R}} = [BeforeImage(\lambda, t)]_{\mathcal{R}}$. We induct on stages after $\mathcal{R}$.

Base step: $\mathcal{S} = \mathcal{R}+1$. At stage $\mathcal{R}$, rule *DoAbort* fires, writing $[BeforeImage(\lambda, t)]_{\mathcal{R}}$ to cache in location λ. By Claim 1,

$$\begin{aligned} [DB(\lambda)]_{\mathcal{S}} &= [BeforeImage(\lambda, t)]_{\mathcal{R}} \\ &= [BeforeImage(\lambda, t)]_{\mathcal{S}} \qquad (by\ Claim\ 5) \end{aligned}$$

Inductive step: $\mathcal{S} > \mathcal{R}+1$. Assume $[DB(\lambda)]_{\mathcal{S}-1} = [BeforeImage(\lambda, t)]_{\mathcal{S}-1}$.

- If $\mathcal{S}-1$ is a read or commit stage, then by Claim 15 and Claim 5, $DB(\lambda)$ and *Log* are unchanged in $[\mathcal{S}-1, \mathcal{S}]$.
- If $\mathcal{S}-1$ is a write stage, then since $\mathcal{S} > [LW(\lambda)]_{\mathcal{S}}$, $\mathcal{S}$ is not a λ-write stage. Rule *DoWrite* fires, writing to cache in a location $\neq \lambda$. By Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{S}-1, \mathcal{S}]$. *DoWrite* logs the write, but by Claim 2, $BeforeImage(\lambda, t)$ is unchanged in $[\mathcal{S}-1, \mathcal{S}]$.
- If $\mathcal{S}-1$ is an abort stage, then by Claim 5, *Log* is unchanged in $[\mathcal{S}-1, \mathcal{S}]$. If $\mathcal{S}-1$ is not an abort-undo stage, then there is no write to cache, so $DB(\lambda)$ is unchanged in $[\mathcal{S}-1, \mathcal{S}]$. If $\mathcal{S}-1$ is an abort-undo stage, then it is not a λ-undo abort stage, since otherwise it would be a (t, λ)-abort-undo stage (by Claim 11), and there has already been a (t, λ)-abort-undo stage $\mathcal{R} < \mathcal{S}$. Then there is a write to cache at a location $\neq \lambda$. By Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{S}-1, \mathcal{S}]$.

In all cases, by the inductive hypothesis, $[DB(\lambda)]_{\mathcal{S}} = [BeforeImage(\lambda, t)]_{\mathcal{S}}$. □.

Claim 18 Let $\mathcal{S}$ be a stage for which $[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}]$ is normal. If $[LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}$, then $[LCV(\lambda)]_{\mathcal{S}} = [DB(\lambda)]_{\mathcal{S}+1}$.

Proof. Assume that $[LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}$. Proof by induction on stages in $\big[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}\big]$.

Base step: $\mathcal{S} = [LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}$. $\mathcal{S}$ is a λ-write stage, so rule *DoWrite* fires, writing $[LCV(\lambda)]_{\mathcal{S}}$ to cache in location λ. Then by Claim 1, $[DB(\lambda)]_{\mathcal{S}+1} = [LCV(\lambda)]_{\mathcal{S}}$, so the proposition holds for $\mathcal{S}$.

Inductive step: $\mathcal{S} > [LW(\lambda)]_{\mathcal{S}}$. Assume the proposition holds for $\mathcal{S} - 1$.

- If $\mathcal{S}$ is a read or commit stage, then by Claim 15, $DB(\lambda)$ is unchanged in $\big[\mathcal{S}, \mathcal{S}+1\big]$, so the proposition holds for $\mathcal{S}$.
- If $\mathcal{S}$ is a write stage, then it is not a λ-write stage since $\mathcal{S} > [LW(\lambda)]_{\mathcal{S}}$. Rule *DoWrite* fires, writing to cache in a location $\neq \lambda$. By Claim 1, $DB(\lambda)$ is unchanged in $\big[\mathcal{S}, \mathcal{S}+1\big]$, so the proposition holds for $\mathcal{S}$.
- If $\mathcal{S}$ is an abort-undo stage, then it is not a λ-undo stage, since otherwise $[LW(\lambda)]_{\mathcal{S}} \neq [LCW(\lambda)]_{\mathcal{S}}$ by Claim 12. Rule *DoAbort* fires and writes to cache in a location $\neq \lambda$. By Claim 1, $DB(\lambda)$ is unchanged in $\big[\mathcal{S}, \mathcal{S}+1\big]$, so the proposition holds for $\mathcal{S}$.
- If $\mathcal{S}$ is an abort stage but not an abort-undo stage, then rule *DoAbort* does not write to cache, so $DB(\lambda)$ is unchanged in $\big[\mathcal{S}, \mathcal{S}+1\big]$; therefore the proposition holds for $\mathcal{S}$. □.

6.6 Claims involving recovery processing

Claim 19 Let $\mathcal{S}$ be an error stage, let $E \in [Log]_{\mathcal{S}}$, and let $p = [EntryPos(E, Log)]_{\mathcal{S}}$. If $\big(\mathcal{S}, \mathcal{S}+p\big]$ is error-free, then $[CurrEntry]_{\mathcal{S}+(p+1)} = E$.

Proof. Similar to that for Claim 13.

Claim 20 If $\mathcal{S}$ is a λ-undo recovery stage, then $[CurrEntry]_{\mathcal{S}} = [LastEntry(\lambda, Log)]_{\mathcal{S}}$.

Proof. Assume that $\mathcal{S}$ is a λ-undo recovery stage but $[CurrEntry]_{\mathcal{S}} \neq [LastEntry(\lambda, Log)]_{\mathcal{S}}$. Let $p = [EntryPos(LastEntry(\lambda, Log), Log)]_{\mathcal{S}}$, and let $q = [EntryPos(CurrEntry, Log)]_{\mathcal{S}}$. Then $q > p$. Let $\mathcal{R} = Err(\mathcal{S}) + (p+1)$. By Claim 19, $[CurrEntry]_{\mathcal{R}} = [LastEntry(\lambda, Log)]_{\mathcal{S}}$. Rule *DoCorrect* fires at $\mathcal{R}$, adding λ to *Corrected*. Since $Err(\mathcal{S}) + (q+1) = \mathcal{S}$, $\mathcal{S} > \mathcal{R}$, so $\lambda \in [Corrected]_{\mathcal{S}}$. This contradicts the assumption that $\mathcal{S}$ is a λ-undo recovery stage. □.

6.7 Lemmata

Our first lemma deals with the time interval between a write to a data item λ and the next error stage following that write (if any occurs). At any stage within this interval, the last committed value can be found at a well-defined place in the log. If the last write to λ is committed, the last committed value of λ is simply the after image of the last λ-entry. If the last write is not committed, the last committed value is the before image of λ with respect to the last writer to λ.

Lemma 1 Let $\mathcal{S}$ be a stage for which $\big[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}\big]$ is normal. Then

$$[LCV(\lambda)]_{\mathcal{S}} = \begin{cases} [NewVal(LastEntry(\lambda, Log))]_{\mathcal{S}+1} & \text{if } [LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}; \\ [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}})]_{\mathcal{S}+1} & \text{otherwise.} \end{cases}$$

Proof. By induction on stages in $\big[[LCW(\lambda)]_{\mathcal{S}}, \mathcal{S}\big]$.

Base step: $\mathcal{S} = [LCW(\lambda)]_{\mathcal{S}}$. $\mathcal{S}$ is a λ-write stage, so rule *DoWrite* fires, logging new value $[LCV(\lambda)]_{\mathcal{S}}$ in location λ. Then by Claim 3, $[NewVal(LastEntry(\lambda, Log))]_{\mathcal{S}+1} = [LCV(\lambda)]_{\mathcal{S}}$. Since $[LW(\lambda)]_{\mathcal{S}} = [LCW(\lambda)]_{\mathcal{S}}$, the proposition holds for $\mathcal{S}$.

Inductive step: $\mathcal{S} > [LCW(\lambda)]_{\mathcal{S}}$. Assume the proposition holds for $\mathcal{S} - 1$.

- If $\mathcal{S}$ is a λ-write stage, then $[LW(\lambda)]_{\mathcal{S}} = \mathcal{S}$ and $[LT(\lambda)]_{\mathcal{S}} = [Trans(Cmd)]_{\mathcal{S}}$.
  - If $[LW(\lambda)]_{\mathcal{S}-1} = [LCW(\lambda)]_{\mathcal{S}}$, then by Claim 18, $[LCV(\lambda)]_{\mathcal{S}} = [DB(\lambda)]_{\mathcal{S}}$.
  - If $[LW(\lambda)]_{\mathcal{S}-1} \neq [LCW(\lambda)]_{\mathcal{S}}$, then by Claim 16, $[LT(\lambda)]_{\mathcal{S}-1}$ aborts before $\mathcal{S} - 1$. By Claim 17,

$$\begin{aligned} [DB(\lambda)]_{\mathcal{S}} &= [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}-1})]_{\mathcal{S}} \\ &= [LCV(\lambda)]_{\mathcal{S}} \qquad (\textit{by inductive hypothesis}) \end{aligned}$$

  Rule *DoWrite* fires, logging $[LT(\lambda)]_{\mathcal{S}}$'s write to λ with old value $[DB(\lambda)]_{\mathcal{S}}$. Then by Claim 2,

$$\begin{aligned} [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}})]_{\mathcal{S}+1} &= [DB(\lambda)]_{\mathcal{S}} \\ &= [LCV(\lambda)]_{\mathcal{S}} \end{aligned}$$

  Since $[LW(\lambda)]_{\mathcal{S}} = \mathcal{S} \neq [LCW(\lambda)]_{\mathcal{S}}$, the proposition holds for $\mathcal{S}$.

- If $\mathcal{S}$ is a write stage but not a λ-write stage, then by Claim 4, $LW(\lambda)$ is unchanged in $\big[\mathcal{S} - 1, \mathcal{S}\big]$. *DoWrite* logs the write, but by Claim 2, $BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}})$ is unchanged in $\big[\mathcal{S}, \mathcal{S} + 1\big]$. Thus the proposition holds for $\mathcal{S}$.

- If $\mathcal{S}$ is not a write stage, then by Claim 4, $LW(\lambda)$ is unchanged in $\big[\mathcal{S} - 1, \mathcal{S}\big]$, and by Claim 5, Log is unchanged in $\big[\mathcal{S}, \mathcal{S} + 1\big]$, so the proposition holds for $\mathcal{S}$.

Our second lemma concerns the interval between a system error and the first stage after the ensuing recovery. At any stage within this interval, if a data item λ has been "corrected" (*i.e.*, undone or redone), the current database value of λ is the value defined in Lemma 1 as the last committed value.

Lemma 2 Let $\mathcal{S}$ be an error stage for which there is a λ-write stage $\leq \mathcal{S}$, and let $\mathcal{T}$ be a stage in $\big(\mathcal{S}, Norm(\mathcal{S})\big]$. Then if $\lambda \in [Corrected]_{\mathcal{T}}$,

$$[DB(\lambda)]_{\mathcal{T}} = \begin{cases} [NewValue(LastEntry(\lambda, Log))]_{\mathcal{S}} & \text{if } [LW(\lambda)]_{\mathcal{S}-1} = [LCW(\lambda)]_{\mathcal{S}-1}; \\ [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}-1})]_{\mathcal{S}} & \text{otherwise.} \end{cases}$$

Proof. By induction on stages in $\big(\mathcal{S}, \mathcal{T}\big]$.

Base step: $\mathcal{T} = \mathcal{S} + 1$. $\mathcal{S}$ is an error stage, so *Restart* fires, updating $Corrected$ so that $[Corrected]_{\mathcal{T}} = \emptyset$. Then $\lambda \notin [Corrected]_{\mathcal{T}}$, so the proposition holds vacuously for $\mathcal{T}$.

Inductive step: $\mathcal{T} > \mathcal{S} + 1$. Assume the proposition holds for $\mathcal{T} - 1$.

- If $\mathcal{T} - 1$ is an error stage, then *Restart* fires, updating $Corrected$ so that $[Corrected]_{\mathcal{T}} = \emptyset$. Then $\lambda \notin [Corrected]_{\mathcal{T}}$, so the proposition holds vacuously for $\mathcal{T}$.

- If $\mathcal{T} - 1$ is a (t, λ)-correcting recovery stage for some t, then by Claim 9,

$$\begin{aligned} t &= [LT(\lambda)]_{\mathcal{T}-1} \\ &= [LT(\lambda)]_{\mathcal{S}-1} \qquad (\textit{by Claim } 4) \end{aligned}$$

– If $[LW(\lambda)]_{\mathcal{S}-1} = [LCW(\lambda)]_{\mathcal{S}-1}$, then by Claim 20,

$$\begin{aligned} [CurrEntry]_{\mathcal{T}-1} &= [LastEntry(\lambda, Log)]_{\mathcal{T}-1} \\ &= [LastEntry(\lambda, Log)]_{\mathcal{S}} \quad (by\ Claim\ 5) \end{aligned}$$

By Claim 7, $t \in [CommitSet]_{\mathcal{T}-1}$. Rule *DoCorrect* fires, writing $[NewVal(CurrEntry)]_{\mathcal{T}+1}$ to cache in location λ. Then $[NewVal(CurrEntry)]_{\mathcal{T}-1} = [NewVal(LastEntry(\lambda, Log))]_{\mathcal{S}}$, so by Claim 1, $[DB(\lambda)]_{\mathcal{T}} = [NewVal(LastEntry(\lambda, Log))]_{\mathcal{S}}$. *DoCorrect* also adds λ to $[Corrected]_{\mathcal{T}-1}$, so $\lambda \in [Corrected]_{\mathcal{T}}$. Thus the proposition holds for $\mathcal{T}$.

– If $[LW(\lambda)]_{\mathcal{S}-1} \neq [LCW(\lambda)]_{\mathcal{S}-1}$, then by Claim 20, $[CurrEntry]_{\mathcal{T}-1} = [LastEntry(\lambda, Log)]_{\mathcal{T}-1}$. Then

$$\begin{aligned} [OldVal(CurrEntry)]_{\mathcal{T}-1} &= [BeforeImage(\lambda, t)]_{\mathcal{T}-1} \\ &= [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}-1})]_{\mathcal{T}-1} \\ &= [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}-1})]_{\mathcal{S}} \quad (by\ Claim\ 5) \end{aligned}$$

By Claim 7, $t \notin [CommitSet]_{\mathcal{T}-1}$. Rule *DoCorrect* fires, writing $[OldVal(CurrEntry)]_{\mathcal{T}-1}$ to cache in location λ. By Claim 1, $[DB(\lambda)]_{\mathcal{T}} = [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}-1})]_{\mathcal{S}}$. *DoCorrect* also adds λ to $[Corrected]_{\mathcal{T}-1}$, so $\lambda \in [Corrected]_{\mathcal{T}}$. Thus the proposition holds for $\mathcal{T}$.

- If $\mathcal{T}-1$ is a correcting recovery stage but not a λ-correcting recovery stage, then rule *DoCorrect* fires, writing to cache in a location $\neq \lambda$. By Claim 1, $DB(\lambda)$ is unchanged in $[\mathcal{T}-1, \mathcal{T}]$. *DoCorrect* does not update *Log* or *Corrected*, so *Corrected* and $BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}})$ are unchanged in $[\mathcal{T}-1, \mathcal{T}]$. Thus the proposition holds for $\mathcal{T}$.

- If $\mathcal{T}-1$ is a recovery stage but not a correcting recovery stage, then rule *DoRecovery* does not fire, so *Corrected*, $DB(\lambda)$ and $BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}})$ are unchanged in $[\mathcal{T}-1, \mathcal{T}]$. Thus the proposition holds for $\mathcal{T}$. □.

Our third lemma states that at the end of a recovery procedure, every data item that has been written to has been corrected.

Lemma 3 If $\mathcal{S}$ is an error stage and there is a λ-write stage $\leq \mathcal{S}$, then $\lambda \in [Corrected]_{Norm(\mathcal{S})}$.

Proof. Assume there is a λ-write stage $\mathcal{R} \leq \mathcal{S}$, but $\lambda \notin [Corrected]_{Norm(\mathcal{S})}$. By Claim 6, there is an entry $E \in [Log]_{\mathcal{S}}$ with $Trans(E) = t$ and $Loc(E) = \lambda$. Let n be the number of entries following E in $[Log]_{\mathcal{S}}$. Then since $[CurrEntry]_{\mathcal{S}+1} = [LogEnd(Log)]_{\mathcal{S}}$ and for each $\mathcal{T} \in (\mathcal{S}, Norm(\mathcal{S}))$, $[CurrEntry]_{\mathcal{T}} = [PrevEntry(CurrEntry, Log)]_{\mathcal{T}-1}$, $[CurrEntry]_{\mathcal{S}+(n+1)} = E$. Rule *DoCorrect* fires at $\mathcal{S}+(n+1)$, adding λ to *Corrected*, so $\lambda \in [Corrected]_{\mathcal{S}+(n+2)}$ and therefore $\lambda \in [Corrected]_{Norm(\mathcal{S})}$, a contradiction. □.

6.8 Theorems: correctness and liveness

Our first theorem proves the correctness of the algorithm: after a system error and recovery, every data item that has been written to has its last committed value stored in the database.

Theorem 1 If $\mathcal{S}$ is an error stage and there is a λ-write stage $\leq \mathcal{S}$, then $[DB(\lambda)]_{Norm(\mathcal{S})} = [LCV(\lambda)]_{\mathcal{S}-1}$.

Proof. Let $\mathcal{R}$ be the first error stage in $[[LW(\lambda)]_{\mathcal{S}}, \mathcal{S}]$. By Lemma 3, $\lambda \in [Corrected]_{Norm(\mathcal{S})}$.

- If $[LW(\lambda)]_{\mathcal{S}-1} = [LCW(\lambda)]_{\mathcal{S}-1}$, then by Lemma 2,

$$\begin{aligned}
[DB(\lambda)]_{Norm(\mathcal{S})} &= [NewVal(LastEntry(\lambda, Log))]_{\mathcal{S}} \\
&= [NewVal(LastEntry(\lambda, Log))]_{\mathcal{Q}} \\
&= [LCV(\lambda)]_{\mathcal{Q}} \quad (by\ Lemma\ 1) \\
&= [LCV(\lambda)]_{\mathcal{S}-1}
\end{aligned}$$

- If $[LW(\lambda)]_{\mathcal{S}-1} \neq [LCW(\lambda)]_{\mathcal{S}-1}$, then by Lemma 2,

$$\begin{aligned}
[DB(\lambda)]_{Norm(\mathcal{S})} &= [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{S}-1})]_{\mathcal{S}} \\
&= [BeforeImage(\lambda, [LT(\lambda)]_{\mathcal{Q}-1})]_{\mathcal{Q}} \quad (by\ Claim\ 5) \\
&= [LCV(\lambda)]_{\mathcal{Q}} \quad (by\ Lemma\ 1) \\
&= [LCV(\lambda)]_{\mathcal{S}-1}
\end{aligned}$$

□.

Finally, we prove the liveness of the algorithm: given a sufficiently long error-free period, a recovery procedure will terminate and normal processing will continue.

Theorem 2 Let $\mathcal{S}$ be an error stage, and let $n = |[Log]_{\mathcal{S}}|$. If there is no error stage in $\big(\mathcal{S}, \mathcal{S}+(n+2)\big]$, then $Norm(\mathcal{S}) = \mathcal{S} + (n+2)$.

Proof. Let E be the entry in $[Log]_{\mathcal{S}}$ such that $[EntryPos(E, Log)]_{\mathcal{S}} = n - 1$; then $[PrevEntry(E, Log)]_{\mathcal{S}+n} = undef$. By Claim 19, $[CurrEntry]_{\mathcal{S}+n} = E$. Rule *ScanLog* fires, updating *CurrEntry* so that $[CurrEntry]_{\mathcal{S}+(n+1)} = [PrevEntry(E, Log)]_{\mathcal{S}+n} = undef$. Rule *ScanLog* fires again, this time updating *InRecovery* so that $[InRecovery]_{\mathcal{S}+(n+2)} = false$. Then since $[Error]_{\mathcal{S}+(n+2)} = false$, $\mathcal{S} + (n+2)$ is normal, so $\mathcal{S} + (n+2) = Norm(\mathcal{S})$. □.

References

[BHG 87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Gray 78] J. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course, Lecture Notes in Computer Science* 60, Springer-Verlag, 1978, 393-481.

[Gur 93] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics," in *Current Trends in Theoretical Computer Science*, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266-292.

[Gur 95] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995.