Logical Logging to Extend Recovery to New Domains

David Lomet
Microsoft Research
Redmond, WA 98052
lomet@microsoft.com

Mark Tuttle
Compaq Cambridge Research Lab
Cambridge, MA 02139
tuttle@crl.dec.com

Abstract

Recovery can be extended to new domains at reduced logging cost by exploiting "logical" log operations. During recovery, a logical log operation may read data values from any recoverable object, not solely from values on the log or from the updated object. Hence, we needn't log these values, a substantial saving. In [8], we developed a redo recovery theory that deals with general log operations and proved that the stable database remains recoverable when it is explained in terms of an installation graph. This graph was used to derive a write graph that determines a flush order for cached objects that ensures that the database remains recoverable. In this paper, we introduce a refined write graph that permits more flexible cache management that flushes smaller sets of objects. Using this write graph, we show how: (i) the cache manager can inject its own operations to break up atomic flush sets; and (ii) the recovery process can avoid redoing operations whose effects aren't needed by exploiting generalized recovery LSNs. These advances permit more cost-effective recovery for, e.g., files and applications.

1 Introduction

Overview

There is a substantial literature of recovery algorithms, and a short literature on explaining recovery. A recent book [6] captures much of this. Recovery algorithms encompass logging, cache management, and recovery itself. Clever algorithms tailored for particular system have been invented, e.g. [2,3], evolving to "physiological" techniques [4] such as ARIES [11]. Much of this has been an attempt to balance the choice of log operation (and hence the logging cost) against the complexity of the cache management needed to keep the database recoverable.

Two papers that described the recovery problem in some generality and characterized recovery methods are [5,1]. In [1], the recovery schemes were classified by the interplay of redo and undo recovery. Closer to our discussion here, [5] classifies recovery algorithms, in part, by their impact on cache management. Two characteristics were whether

1. the entire cache needed to be flushed atomically (ATOMIC vs. ~ATOMIC) to the stable state and
2. a cached object could be removed from the cache at any time to permit its portion of the cache to be reused by another object (STEAL vs. ~STEAL).

The physiological recovery methods [4,11] have become the methods of choice in part because they are (~ATOMIC, STEAL), hence maximizing cache management flexibility, while also providing high concurrency. Physiological operations are of the form $X \leftarrow f(X)$, i.e. they transform a single recoverable object. For example, inserting a new record on a page, where only the new record need be logged, transforms the page from a state without the record to one with the record.

In [8], we introduced a general framework for understanding redo recovery. Two factors motivated us. First, we wanted to explain existing recovery technology. Second, we wanted to generalize recovery technology to enable its exploitation in domains other than database recovery. We showed how to exploit more powerful log operations in [7] to reduce recovery cost for applications. Our focus here is on redo recovery using logical operations that permit cost-effective recovery for new areas like applications and file systems. For this, we need to understand how the choice of log operations impacts logging cost, cache management, and recovery itself.

Recovery Opportunities and Problems

It is usually desirable to choose log operations that minimize logging overhead during normal operation. Figure 1 illustrates how, by logging "logical" operations, to avoid writing large data values to the log and contrasts this with the logging cost of "physiological" log operations. In Figure 1(a), logical operation **A** $(Y \leftarrow f(X,Y))$ reads objects X and Y and writes object Y. Then operation **B** $(X \leftarrow g(Y))$ reads Y and writes X. These "logical" operations can be very efficiently logged. The log record for **A** (shown next to ***LOG:***) indicates that X and Y have been read by making them arguments to f. Further, Y has been written (it precedes the colon, and operation **A** (i.e. function f) did the transformation. In Figure 1(b), we describe the changes to X and Y with "physiological" operations. These operations must involve only a single object. Thus, when we now log the write of Y during operation **A**, this requires that we write either the value of X or the result of $f(X,Y)$ in the log record for operation **A**. In the former case, our log record for **A** still indicates that Y has been read and written, and that f was the function that did the transformation. Now, however, we see that the value of X at the time of the operation has been logged. This is denoted by ***log(X)***. This logged value now becomes input to the logged operation **A** on the ***LOG*** in Figure 1(b). Similar considerations apply to operation **B**.

In database systems, updates are typically to records on a page, so it is convenient to formulate updates as physiological operations updating pages. Databases latch small objects like pages in order to facilitate concurrency. In other domains such as application recovery, however, concurrency is often less of an issue. Application designers may program in terms of updates to much larger objects. Here, logging logical operations on large objects to avoid logging the objects themselves can result in enormous savings. Logging logical operations, however, can have a profound impact on cache management and recovery. The goal of this paper is to reduce this impact.

We give some examples of how logical logging can substantially reduce the logging required during normal execution.

Application Recovery: In [7], we provided logical operations for recovering application state. Relating these to operations A and B of Figure 1, *Y* plays the role of application state and *X* the object that is being read and written

- Application read (*R(X,Y)*): Application *Y* reads object *X* into its input buffer, transforming the state of *Y* to a new state *Y′*. This application read operation has the form of operation **A**. Using a logical log operation avoids the need to log the value of *X* that is read or the value of *Y* that is written.
- Application write ($W_L(Y,X)$): Application *Y* writes object *X* from its output buffer. This does not change application state *Y*. This "logical" application write operation has the form of operation **B.** Logical writes were not included in [7] because of the potential for cyclic flush dependencies.
- Application execution (*Ex(Appl)*): An application's execution between database calls is a physiological operation *Appl = Ex(Appl)*. The operation begins when control is returned to *Appl*, its execution transforms *Appl*'s state to the new state when *Appl* next calls the database system. Parameters for *Ex(Appl)* are stored in the log record

Physiological operations cannot achieve the log space savings for application reads and writes, as in both cases, the values read and/or the values written would need to be logged.

File System Recovery: The forms of logical operation described in Figure 1 can also provide recovery for a file system. An operation that copies file X to file Y is in the form of operation **B**. This same form describes a sort, where X is the unsorted input and Y is the sorted output. In neither case do we log the values of input or output files. Only the transformations are logged and the source and target files id's. Were physiological operations used, we would need to log file *Y* (or file *X*) in its entirety.

Database Recovery: Logical logging is also useful in database recovery. Operations of the form of operation **B** of Figure 1(a) can be used in B-tree splits, i.e., to copy half the contents of a full B-tree page to a new page. *X* denotes the old page and *Y* the new page. The split operation moves keys greater than the split key to the new page. A logical split operation avoids the need to log the contents of the new B-tree node, which is required when using the simpler physiological operation for **B** of Figure 1(b).

The key to logging economy from using logical operations in new recovery domains is logging the identifiers of the sources of data values instead of the values. Since many operations have large operands, page size or larger, logging a source identifier that is unlikely to be larger than 16 bytes is a great saving. Both application state and files may be many pages in size.

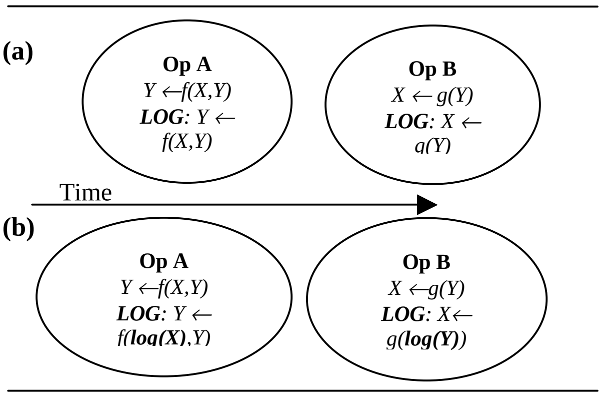
(a)
Op A
$Y \leftarrow f(X,Y)$
LOG: $Y \leftarrow f(X,Y)$
Op B
$X \leftarrow g(Y)$
LOG: $X \leftarrow g(Y)$
Time
(b)
Op A
$Y \leftarrow f(X,Y)$
LOG: $Y \leftarrow f(\mathbf{log(X)},Y)$
Op B
$X \leftarrow g(Y)$
LOG: $X \leftarrow g(\mathbf{log(Y)})$

Figure 1: Logical (a) and physiological (b) operations needed to accomplish the same result for operations A and B.

Our obligation when taking advantage of the logging economy of logical log operations is to ensure that the values of the data sources needed during recovery are the same as were present during the original execution. This complicates cache management and is the price we pay for logging efficiency. Thus, in our example of Figure 1(a), once **A** is executed, a flush order dependency exists to ensure that **A**'s result *Y* is flushed prior to any subsequent change to *X* being flushed. If an updated *X* were flushed first, we could not replay **A** to regenerate the correct value for *Y* were the system to crash and require recovery.

Flush order dependencies can require the atomic flush of multiple objects. For example, atomic flushing (propagating) was a requirement of IBM's System R [3]. While we might restrict operations to write only one object (but allow reading other objects), this does not avoid the problem. In Figure 1(a), once **B** is executed, and before its output *X* is flushed, we must prevent any modification to **B**'s input *Y* in order for **B** to be *replayable.* However, operation **A** requires that Y be flushed before *X*. Only flushing *X* and *Y* together atomically satisfies both dependencies, because then there is no need to replay one operation after the other 's updates have been flushed.

Media recovery using database backups is a further complication. A backup needs to be recoverable just as the stable database is in order for media recovery to succeed. However, backups are usually asynchronous with normal execution (a fuzzy backup). Copying the database to the backup can introduce flush order violations for the backup even when cache management honors flush order for the stable database. We describe how such backups can be kept recoverable in [10].

Paper's Contribution

Our goal is to show that recovery based on log operations like those in Figure 1(a) is very advantageous. The recovery system can choose from a wider class of operations, logical operations as well as physiological. It can decide whether the cost of writing large objects to the log should be the principal concern, hence justifying logical operations despite more difficult cache management. It might also decide that the objects involved are not so large as to justify more complex cache management, and hence choose physiological operations. ***The wider choice of forms of log operations permits cost effective recovery for new domains, where recoverable objects can be much larger than is usually the case for database systems.***

Our focus is on the cache management and recovery costs and complications resulting from logical log operations. The paper introduces a fundamental new insight in how to install operations that permits more flexible cache management. It then introduces two additional innovations exploiting this so that (i) the cache manager can cope with logical operations, and (ii) recovery processing can be optimized.

New Write Graph

Flush dependencies are captured in a *write graph* [8], defined in Section 3. Objects associated with a write graph node must be flushed atomically. A well formed write graph is acyclic to impose a flush order on cached objects. When a cycle arises, we "collapse" the nodes involved into a single node which will have multiple objects associated with it. The I/O overhead and system interruption of multi-object atomic flushing makes this costly.

Restricting logged operations can ensure that it is unnecessary to flush more than one object at a time. Physiological operations avoid all flush dependencies. In [7], application operations update only one object and logical log operations only describe application reads. This prevents write graph cycles, but require application write operations be logged as physical operations. So logging cost is low for application reads, but an application write log record includes the data value to be written. This is costly when objects are large, e.g. files.

With the write graph of [8], if the cache manager ever determines that two objects *x* and *y* are associated with one write graph node, then they must be flushed atomically when they are next written to disk. In this paper, we show how subsequent updates make it possible to relax this atomicity and flush them separately if they are flushed in the right order. We capture this relaxation in terms of a new *refined write graph rW*. The fundamental insight is that subsequent updates can cause the value of *x* to become unexposed (i.e., subsequent updates do not need to read *x* to compute its next value) and not have to be flushed

Innovations

Cache Manager Initiated Writes

The cache manager (CM) can introduce operations that break up atomic flush sets by making object values unexposed, so that they do not need to be flushed to install earlier operations. These "identity" operations do not change the objects involved but result in logging activity that makes separate flushing possible. Each operation is a physical write that saves the value (state) of an object in a log record. This has a lower cost and less impact on system operation than atomic flushing. The result is that the recovery system can permit logical operations that are flexible and efficient without fear that the overhead of atomic flushing will undermine system performance or concurrency.

Generalized Recovery LSNs

With *rW*, it is sometimes possible to reduce the number of operations that require redo recovery without flushing all their updated objects to the stable database during normal execution. This can be exploited to optimize recovery, which is especially important when the logged operations are expensive to re-execute, e.g., applications [7]. This is accomplished by using recovery LSNs (*rLSNs*) [11] in our more general setting with logical operations, and based on how we install their effects. An object's *rLSN* identifies the earliest log record needed for its recovery. Logical operations can make determining the *rLSN* rather subtle.

Paper Organization

In the next section, we present a simplified version of the recovery framework of [8] to reduce the framework description to manageable size. We define installation graph, exposed objects, explainable database state, and recoverable database. We show how log-based recovery systems can be understood in terms of these notions. The definitions and theorems have been extended from what was in [8] to clarify the definition of exposed objects, which we exploit in the current paper.

Section 3 discusses how to translate from the installation graph ordering operations to a write graph ordering object flushes. We then introduce our new write graph *rW* that improves on the write graph *W* of [8] by exploiting the fact that unexposed objects need not be in an understandable state for recovery to succeed. This lets us remove objects from atomic flush sets, and ultimately to flush objects one at a time.

We discuss how the CM can exploit *rW* to break up atomic flush sets in section 4. It does this by introducing operations of its own (identity writes) and interjecting them into the stream of operations that are recoverable. We compare this to the logging cost and concurrency difficulties of a multi-object atomic flush.

A side effect of the refined write graph is that we can sometimes "install" operations without flushing their results to the stable database. We can exploit this to optimize recovery processing, shortening the recovery log scanned and reducing the number of operations that need to be redone. We describe this in section 5.

Finally, we summarize the results of our efforts in section 6.

2 Recovery Framework

Installation is the fundamental recovery concept. Pragmatically, an *installed* operation is one that no longer needs to be re-executed in order to recover the stable database state. An operation is usually installed by writing to the stable state (disk) the objects it changes to include the operation's effects in the stable state. We call the writing of changed objects to stable state *flushing*.

We simplify the recovery framework of [8] by collapsing the notions of *update*, *redo*, and *operation* into the single notion of *operation*. The original framework permitted an operation to consist of several updates, each atomically installed. Here, we restrict operations to a single update. Note, however, that while we simplify the description, the techniques should nonetheless be applicable in the fully general setting.

We make common assumptions about the recovery system. The write-ahead log (WAL) protocol is followed in which all changes in stable system state must be described by operations on the stable log before the changes caused by the operation are "installed". Further, we assume that the redo recovery process is ARIES-like in that we “repeat history”. Operations on the log are assumed to be in conflict order. Note that we do not require that operations on the log be totally ordered, as conflict order is not a total order.

Installation Graph

The basis for the recovery framework is an *installation graph*. It constrains the order in which changes made by operations can be made part of the stable database state, and provides a way of *explaining* what operations can be considered installed. This graph uses edges to order operations, like the conflict graph, but installation ordering is much weaker. If, after crash, the state can be explained in terms of this graph, then we can recover by selectively "replaying" logged operations. The installation graph restricts when objects can be *flushed*. It lets us construct a *cache management algorithm* that guarantees that the stable state remains explainable, and a *recovery algorithm* that recovers any explainable state. Thus, it captures the impact of the choice of logged operations on the recovery process.

Our *installation graph* is a graph whose nodes are operations and whose edges constrain the order in which the operations are installed. It is obtained from the conflict graph by keeping all read-write edges, throwing away all write-read edges, and keeping only some write-write edges. We thus define the *installation graph* for a history H (or conflict graph) with operations $\boldsymbol{H}$ as a directed graph where each node is labeled with an operation O. An operation O is characterized by the objects it reads (*readset(O)*) and the objects it writes (*writeset(O)*). For distinct operations O and P, there is an edge from O to P if $O < P$ in H and either

1. *read-write edges*: $readset(O) \cap writeset(P) \neq \varphi$.
   Here, a later operation updates an object read by an earlier operation. If *P’s* updates are posted to the stable database, but a crash prevents *O’s* from being posted, then O must be replayed to produce the missing effects, but cannot because *readset(O)* has changed. Hence, the database will not be recoverable.
2. *write-write edges*: P is in *must(O)* but not in *can(O)*. The operations of *must(O)* must be recovered by re-execution should *writeset(O)* be reset by redoing O, because some of their updates will be reset. The operations of *can(O)* are those that can be recovered as a result of recovering only operations in *must(O)*. See [8] for precise definitions. Write-write order is usually not violated during normal execution as we only write the latest state, but it can pose difficulties during recovery. Recovery deals with incomplete information. Whether a reset violates write-write order may be unknown. One strategy is to log only operations for which *write-write* edges do not arise. A second strategy, which we pursue here, never resets state during recovery, and hence *write-write* order will not be violated.

Explainable States

For recovery to be feasible, we must understand the stable system state after a crash in terms of what operations have their effects already present (installed) in the stable system state and what operations do not (uninstalled). During recovery, certain important objects must contain correct values. We define a set $\boldsymbol{I}$ to be a prefix set if for every operation O in I, $\boldsymbol{I}$ contains every $P < O$ (in installation order) of $\boldsymbol{H}$. An object x to be *exposed* by a prefix set I iff one of the following conditions is true:

1. no operation in $\boldsymbol{H} - \boldsymbol{I}$ reads or writes x, or
2. some operation in $\boldsymbol{H} - \boldsymbol{I}$ reads or writes x, and the minimal operation (earliest in conflict order) reads x.

A prefix set $\boldsymbol{I}$ of operations in $\boldsymbol{H}$ *explains* state S if for every object x exposed by I, the value of x in S is the value of x after the last operation (in conflict order) of I. We call the operations in $\boldsymbol{I}$ installed operations, while the operations in $\boldsymbol{H} - \boldsymbol{I}$ are uninstalled operations (for this explanation). Thus, if S is the state of the stable database at the start of recovery, then the *sequence* of operations that created S is unimportant. Only the *set* of operations considered installed in S is important. This is crucial, as it is impossible in general to determine the exact installation sequence that leads to a database state. Indeed, it may be possible to explain S with several different sets of installed operations.

Minimal Uninstalled Operations

An operation O is *applicable* to a state S at recovery time if for every object x in *readset*(O), the value of x in S is given by O's before image. This means that O reads the same values during recovery as it did during normal operation, so it will write the same values as well. We can install O by setting every object x in *writeset(O)* to its after image produced by O. An operation O is *installable* in a state S if the database state $S' = SO$ obtained by installing O in S is explainable by a prefix of the installation graph. We can extend these definitions to sequences of operations in the obvious way. A prefix set $\boldsymbol{I}$ can be *extended* by an operation O if there is no *write-write* edge from O to any operation in I and $\boldsymbol{I}$ contains all $P < O$. We define $extend(I,O)$ as deleting every operation in *must*(O) from I, and then including O. This is the result of removing all operations that O de-installs, and then making sure that O's effects are present.

An explainable state can be recovered by installing uninstalled operations in installation order, starting from a minimal uninstalled operation, i.e., one with no uninstalled installation predecessors. Thus, we have:

Theorem 1: Let S be a state explainable by a prefix set $\boldsymbol{I}$. If O is a minimal uninstalled operation of $\boldsymbol{H} - \boldsymbol{I}$, then O is applicable to S and $extend(I,O)$ explains SO, so O is installable in S.

System Recovery

Theorem 1 suggests how to recover an explainable state: choose a minimal uninstalled operation *O*, install it, and repeat. Of course, *O* must be on the log. Informally, a stable state is recoverable if it is explainable and the log contains the uninstalled operations. A database ***D*** consists of stable state ***S***, a log ***L***, and a history ***H***. It is *explainable* if there is a prefix set ***I*** of the history ***H*** such that ***I*** explains ***S*** and operations in ***H* - *I*** are on the log ***L***.

Procedure *Recover(D,I)* in Figure 2 recovers a database ***D*** explained by a *I*. The algorithm considers all operations *O* in log order, and then invokes a test **REDO**(*D*,*I*,*O*) to determine whether *O* should be installed into the state ***S*** of database ***D***. The recovery task of the CM is to guarantee that there is always at least one ***I*** that explains ***S***. The task of *Recover(D,I)* is to ensure that operations replayed during recovery preserve at least one ***I*** that explains the recovering state ***S***.

REDO(*D*,*I*,*O)* must return true if *O* is a minimal uninstalled operation of ***I***. In this case, Theorem 1 guarantees that *O* is applicable and installable. It can be hard to tell that an operation is uninstalled in ***I***, so a recovery method may end up redoing operations already installed in ***I***. In fact, whenever *O* is applicable and installable, it is okay for the test to return true. Hence, **REDO**(*D*,*I*,*O*) satisfies:

- *Safety*: If it returns true, then
    - *O* is applicable to *S*, and
    - there are no write-write edges from *O* to operations in ***I***
- *Liveness*: It returns true if *O* is a minimal uninstalled operation of *I*

No recovery algorithm actually maintains *I*. However, *I* is only used when evaluating **REDO**(*D*,*I*,O). Cache management will guarantee that there is at least one explaining *I* at the time of a crash. We need to design a satisfactory **REDO**. We can then prove that the invariant "*D* is explained by *I* " holds after each step of *Recover(D,I)*, and conclude that an explainable database is recoverable. Of course, designing an effective **REDO** test and proving that the test satisfies the preceding requirements based on the explanations that it identifies is non-trivial.

Theorem 2: If database *D* is explained by *I*, then *Recover(D,I)* is an idempotent recovery process that recovers *D*.

```
procedure Recover(D,I)
        while the log L is not empty do
                choose a minimal operation O in the log L
                if REDO(D,I, O) then
                        execute (redo) O
                        flush writeset(O) to S,
                        replace I with extend(I,O)
                        delete O from the log L
                        end then
                end do
        end proc
```

Figure 2: Recovering a database *D* explained by prefix sets of installed operations contained in Σ_I.

So the recovery task is clear. We need a CM that guarantees there is an ***I*** that explains the stable state *S*. And we need a recovery process that can correctly identify an ***I*** that explains *S* so that as the log is scanned, the **REDO** test causes the execution of the appropriate operations. Section 3 addresses cache management, section 4 the **REDO** test.

3 Write Graphs

Background

A cache manager divides volatile state into a ‘‘dirty” part and a ‘‘clean” part (not discussed here). An object enters the dirty volatile state when an operation updates it, can be the subject of multiple updates while there, and leaves the dirty volatile state upon being flushed to the stable database. Traditionally, a database CM recovers database pages and tracks cached pages in a page table. We abstract that to an *object table* as we apply recovery to more than just pages. Objects of the dirty volatile state are written to the stable database for two reasons. First, the volatile state can be (nearly) full, requiring that objects currently present be removed to make room for new objects. Second, it may be desired to shorten recovery by checkpointing, i.e. truncating the stable log. Only installed operations can be removed from the log, so it may be necessary to install some operations before log truncation. Systematic installation permits a prefix of the log to be truncated while preserving stable system state recoverability, which requires that the log contain all operations that are uninstalled in some explanation of the stable state.

The duty of the CM is to ensure that there is always at least one ***I*** that explains the database ***D***. Thus, during normal operation, the CM can focus on the ***I*** that is the "leading edge" set of installed operations, and ensure that this set will always explain ***D***. (The situation is more complicated during recovery.)

The CM’s central problem is that installation graph nodes are *operations* but the CM writes *objects*. The CM must write objects so that operation atomicity and installation order are honored. It computes a *write graph* for this purpose. Each write graph node *v* has an associated set *ops(v)* of uninstalled operations that are maintained as part of volatile state and a set *vars(v)* of the objects (variables) these operations write. There is an edge from *v* to *w* in the write graph if there is an installation edge from any operation *P* in *ops(v)* to any operation Q in *ops(w)*.

The operations of *ops(v)* are installed by flushing the last values written to the objects of *vars(v)* and *vars(v)* is written atomically in order to guarantee operation atomicity and installation order within *v*. The *vars(v)* sets must be flushed in write graph order to guarantee installation order. (Physiological operations result in a degenerate write graph, each node of which is associated with the operations that write to a single object, and with no edges between nodes and hence with no restrictions on flush order.)

The installation graph is the gold standard for explaining and controlling installation of operations. Write graphs are derived from the installation graph to expedite cache management. Many write graphs are possible. For example, a single node write graph that requires atomically flushing the entire cache is sufficient to ensure recovery (though it is rarely necessary). It should be clear that the more “refined” the write graph is (i.e., the more nodes into which the uninstalled operations can be divided) the more flexibility the cache manager will have. The write graph *W* of [8], which we describe next, is also sufficient and will usually consist of more than one node. It too, however, is often not necessary.

```
procedure WriteGraph(In)
  T ← the transitive closure of O ~ P
    iff writeset(O) ∩writeset(P) ≠ φ for operations labeling
            nodes of In
  V ← collapse In with respect to the equivalence classes of T
  S ← the strongly connected components of V
  W ← collapse V with respect to the equivalence classes of
            nodes in S
  return(W)   /* collapsing V made W acyclic */
  end proc
```

Figure 3: Computing the write graph *W*.

```
procedure PurgeCache
   compute the write graph W from the operations in the cache
   choose a minimal node v in W
   if operations in ops(v) are not already on the stable log then
      write a conflict graph prefix of operations in the volatile
         log buffer that include ops(v)  to the stable log in conflict
         order (WAL protocol)  (this adds the operations to the
         stable history)
      delete the conflict graph prefix of operations from the
        volatile log buffer that were written
      end then
  atomically write values of objects in vars(v) to the stable
     state
  remove v from W
  delete objects in vars(v) from the dirty cache
  delete ops(v) from the cache
  return
  end proc
```

Figure 4. The cache purging algorithm *PurgeCache*.

W is computed by *WriteGraph(In)* from the subgraph *In* of the installation graph determined by operations in the cache that are uninstalled in the stable database (operations in ***H - I*** for the set ***I*** that explains the stable state). This algorithm uses the idea of collapsing a graph *A* with respect to a partition *Π* of its nodes. Each set of the partition represents objects that must be written atomically. The result is a graph *B* where each node *w* corresponds to a class π_w in the partition *Π*. An edge exists between nodes *v* and *w* of *B* if there is an edge between nodes *a* and *b* of *A* contained respectively in π_v and π_w. This idea is used twice in computing the write graph, once to collapse intersecting updates, and again to make the write graph acyclic. Only acyclic write graphs specify a feasible flush order. In [7] we showed how to *dynamically* compute *W* for the application operations described there. Here we describe the computation of a complete *W* from the set of uninstalled operations maintained in a cache.

When the CM uses an algorithm equivalent to ***PurgeCache*** in Figure 4 to write to the stable state during normal execution, the recoverability of the stable database is preserved. To prove this requires that we show that for stable state ***S*** explainable by ***I,*** that invariant *Inv* is preserved, where *Inv* is defined to be "*Inv(**I**)* is rue for some ***I*”.**

Invariant *Inv(**I**)*: for every operation *O* in the cache:

1. there are no write-write edges in the volatile history’s installation graph from *O* to operations in ***I,*** and
2. every operation *P* < *O* (in conflict order) is in *I* or in the cache.
3. If operation *P* is in the cache and *Q* will be reset by installing *O*, i.e. *Q* is in *must*(*O*), and *Q* < *P* in the installation graph for the volatile database, then *Q* is in the cache and there is a path of write-write edges from *O* to *Q* in the write graph.

The first two conditions let us prove that ***I*** can be extended by *O* when we finally install *O* in the stable database. The last condition ensures that if installing *O* deletes an operation *Q* from ***I***, and hence could possibly violate the second condition requiring that all *Q* < *P* are in ***I*** or in the cache for all *P* in the cache, then *Q* must be in the cache (and hence will be (re)installed with or after *O*). When *O* is a minimal uninstalled operation in the volatile state, these conditions imply that ***I*** can be extended by *O*. So Theorem 1 says that installing *O* in the stable state will yield an explainable state.

Three lemmas (given later when discussing rW, and with proofs in the appendix) show that ***PurgeCache***, both during normal operation and during recovery, correctly installs operations when *Inv* holds, and that its execution preserves invariant *<u>Inv</u>*. We thus have:

Theorem 3: *PurgeCache* preserves the invariant *Inv* and hence the recoverability of the stable database.

Capturing a More Precise Flush Ordering

Consider Figure 5 where *X* and *Y* need to be flushed together atomically in *W* after operation A. Despite this, after operation B, we can safely flush Y when no uninstalled operations read the value of *X* written by A. We recover *X* by replaying operation B, whose log record tells how to recreate it from the value flushed to *Y*. Hence, recovering *X* does not depend on the replay of A. We don’t care what value A wrote to *X* because it isn’t read by subsequent uninstalled operations; i.e. it is *unexposed*.

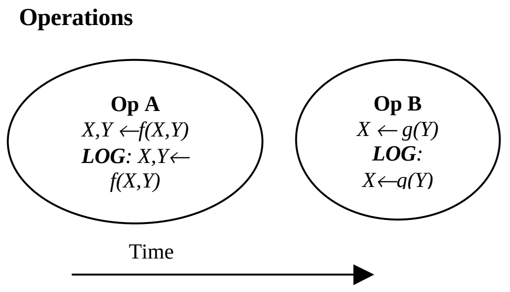

Figure 5: An example showing that a more precise flush order is possible than what is captured in *W*.

For a node *n* of *W*, |*vars(n)*| is monotonically increasing, resulting in ever larger atomic flushes, until *vars(n)* is finally flushed. We can do better by exploiting objects that are *not exposed*. We alter the initial transitive closure that generates *T* (see Figure 3) to construct a refined write graph *rW*. The earlier construction of *T* lost information that can keep nodes in *rW* from coalescing, resulting in more nodes with fewer objects per node.

To ease exposition, we introduce some notation (Table 1 has the complete set). The set of objects written by operations in *ops(n)* of a write graph node *n* is *Writes(n)*, the set read is *Reads(n)*. The value written by the last update to object *X* by an operation in *ops(n)* is *Lastw(n,X)*. *X* will be exposed should a subsequent operation in another write graph node read *Lastw(n,X)*.

Two salient differences between *W* and *rW* are:

- In *W*, *vars(n)* = *Writes(n)*. In *rW*, not all objects in *Writes(n)* need be in *vars(n)*. Nonetheless, we install all operations in *ops(n)* by flushing *vars(n)*.
- There may be extra edges between nodes *n* and *m* in *rW* to ensure that certain objects are not exposed or that operations in *must(O)* always follow *O* in the write graph, and these need not be installation graph edges between operations in *ops(n)* and *ops(m)* (required for graph *W*).

Figure 6 shows how *rW* is incrementally constructed from the uninstalled operations by including them in conflict order. When presented with a new operation *Op*, we must assign it to a node *m* of *rW*. Node *m* may be new or be formed by merging nodes *n* whose exposed updated objects (*vars(n)*) overlap with objects both written and read by *Op*., i.e. *exp(Op)*. Objects in *notexp(Op)* are not exposed immediately before *Op* executes. Objects in *exp(Op)* have updates that depend on their previous values and hence are unavoidably "exposed". They must be flushed at the same time as objects written by operations that updated them earlier. Each *X* in *writeset(Op)* becomes a member of *vars(m)*.

Operations	
Ex(A)	Application Execute: reads and writes *A*
R(A,X)	Application Read: reads *A*,*X* and writes *A*
$W_P(X,v)$	Application Physical Write: writes *X* with *v*
$W_{PL}(X)$	Application Physiological Write: reads and writes *X*
$W_L(A,X)$	Application Logical Write: reads *A*, writes *X*
$W_{IP}(X,val(X))$	CM Identity Write of *X* with its current value
Operation	***Op* Attributes**
readset(Op)	Read set of operation *Op*
writeset(Op)	Write set of operation *Op*
notexp(Op)	‘‘not exposed" objects of *Op* = *(writeset(Op) – readset(Op))*
exp(Op)	"exposed" objects of *Op* = *(writeset(Op) ∩ readset(Op))*
Write Graph	**Node *n* Attributes**
ops(n)	Set of operations associated with node *n*
vars(n)	Subset of *Writes(n)* flushed to install *ops(n)*
Reads(n)	∪ {*readset(Op)*| *Op* in *ops(n)*}
Writes(n)	∪ {*writeset(Op)*| *Op* in *ops(n)*}
Notx(n)	*(Writes(n) – vars(n))*, the "not exposed" objects of *n*
Lastw(n,X)	Last value or *SI* of *X* written by an operation of *ops(n)*
W	Write graph of [8]
rW	Refined write graph

Table 1: Introduced notation

Only objects in *notexp(Op)* can also occur in *vars(p)* for any other node *p* and we remove them. Thus, each *X* is a member of only one *vars(p)* for all *p*. These "blindly" updated objects can now be recovered independently of their earlier values. Note that there may now be operations in *ops(p)* that have *write-write* conflicts with *Op*. Hence, *rW* may have *write-write* edges to ensure that operations in *must(op)* are installed after *op* is installed, for every *op*. This preserves part three of the cache invariant *Inv(**I**)*.

We must ensure that objects *X* in *Notx(p)* are not exposed when we flush *vars(p)* to install *ops(p)*. The value *Lastw(p,X)* for *X* in *Notx(p)* is not needed for the subsequent update of *X* but might be needed by operations in another node *q* that has operations that read *X*. For this, we define edges to *p* from each node *q* with an operation in *ops(q)* that reads *Lastw(p,X)*. In *rW*, we call this edge an *inverse write-read* edge because it is from a later reader to an earlier writer. Hence, objects in *Notx(p)* need not be preserved by flushing as any operation that depends on these variables has already been installed before *p* is installed.

Procedure *addop_rW* is invoked as operations arrive at the CM. This procedure is an approximate substitute for the first collapse in the procedure of Figure 3 for constructing *W*. Should cycles arise as a result of *addop_rW*, we exploit a second collapse, as was done for the construction of *W*, to make *rW* acyclic.

Since we now have an incremental construction for *rW*, we need to also make explicit how operations are removed from *rW*. A node *n* of *rW* is removed when *ops(n)* are installed via flushing the objects in *vars(n)* when *n* has no *rW* predecessors. Removal of *n* removes all its edges. This never results in new cycles. Hence, we can remove a node from *rW* with *W*'s ***PurgeCache***.

```
procedure  addop_rW(rW,Op))
  /* the incremental version of  the first collapse for rW  */
  create m in rW by merging nodes n in rW for which
              $vars(n) \cap exp(Op) \neq \varphi$
    $ops(m) = ops(n) \cup \{Op\}$
    $vars(m) = vars(n) \cup writeset(Op)$
    $edges(m) = \{edges(n)\} \cup$ /* new read-write edges */
                $\{<p,m> \mid Reads(p) \cap writeset(Op) \neq \varphi\}$
for  each $p \neq m$ of rW  with $vars(p) \cap notexp(Op) \neq \varphi$  do
     /* remove "not exposed" objects from vars of other nodes */
      $vars(p) = vars(p) - notexp(Op)$
      $Notx(p) = Writes(p) - vars(p)$
      /* Op in must(op) for an op in ops(p), a write-write conflict  */
       include edge <p,m> in rW
      for each node $q \neq p$ of rW do
       if $op \in ops(q)$ reads $Lastw(p,X)$, $X \in (Notx(p) \cap notexp(Op))$
         then
          /* include "reverse" write-read edge to ensure that Notx(p)
             is not exposed when node p has no predecessors */
          include edge <q,p> in rW
          end then
        end do
      end do
    end proc
```

Figure 6: Computing the refined write graph incrementally using *addop_rW*.

The correctness of *rW* follows from three lemmas that we alluded to before when discussing *W*. We state the lemmas here. Their proofs, which apply equally to *W* and *rW*, are given in [9].

Lemma 1: Suppose the database satisfies *Inv(I)*. If *v* is a minimal node of the write graph, then *I* can be extended by any ordering *T* of the operations in *ops(v)* consistent with the conflict ordering, and hence *extend(I,T)* is defined.

Lemma 2: Suppose the database satisfies *Inv(I)*, *which* implies that *I* explains the stable database. If *v* is a minimal node of the write graph and *T* is an ordering of the operations in *ops(v)* consistent with conflict order, then *extend(I,T)* explains the stable database obtained by writing the objects in *vars(v)* to the stable database.

From these two lemmas, we know that the new state of the stable database is explained by the operations that we have installed via flushing *vars(v)*. It remains to be established that the invariant is preserved, which essentially tells us that what is in the cache is acceptable for continuing recovery going forward.

Lemma 3: Suppose the database satisfies *Inv(I)*. Let *v* be a minimal node of the write graph and *T* be some ordering of the operations in *ops(v)* consistent with the conflict ordering, and let *I'* = *extend(I,T)*. Then the database obtained by writing the objects in *vars(v)* to the stable database satisfies *Inv(I')*.

Then ***PurgeCache*** will, when using *rW*, keep the stable database recoverable because Theorem 3 is true for *rW* as well as for *W*.

4 Cache Management

Multi-object Flush Sets

There is no guarantee that $|vars(n)| = 1$ for *n* in *rW*. As seen in Figure 7, multiple objects written by one operation, at least temporarily, are in one atomic flush set. Node *1* of *rW* initially has a flush set $vars(1) = \{X,Y\}$. Only after operation C is $|vars(1)| = 1$. For *W*, C is always added to *ops(1)*.

Even when all operations write only single objects, cycles can arise in *rW*. Consider the sequence *(a)* $Y = f(X,Y)$; *(b)* $X = g(Y)$, *(c)* $Y = h(Y)$. Operations *(a)* and *(b)* initially are in separate *rW* nodes with *Y* preceding *X* in flush order. When operation *(c)* updates *Y*, *X* must be flushed before *Y* with its new value is flushed. Thus, a cycle involving *rW* nodes with objects *X* and *Y* has formed. Cycles are collapsed into a single node, bringing together objects, previously in separate flush sets, into a multi-object flush set for the resulting node. Note that operation *(a)* has the form of an application read, *(b)* of an application write, and *(c)* of an application execute. Hence, these application recovery operations can potentially lead to cycles that collapse to nodes with multi-object flush sets.

Cycles arise even more often in write graph *W*. The CM in [7] did not have to deal with cycles in *W* because we precluded logical write operations like operation *(b)*. Instead of logical writes, we introduced physical writes $X = g(logged(Y))$, where the value for *X* is read from the log record. With that restriction, no cycles arose. This was fortunate, as *W*'s atomic writes sets never shrink.

As indicated in the example of Figure 7, and unlike with *W*, a subsequent operation may remove objects from a multi-object atomic flush set in *rW*, reducing it in some cases to a single object. Two questions arise.

1. How do we know that such an operation will be forthcoming? Operations have their origin in applications that are outside of the control of the recovery system.
2. How long do we wait for such an operation? Even if such an operation *eventually* arrives, as with temporary files being deleted, effective cache management is all but impossible.

This is highly unsatisfactory.

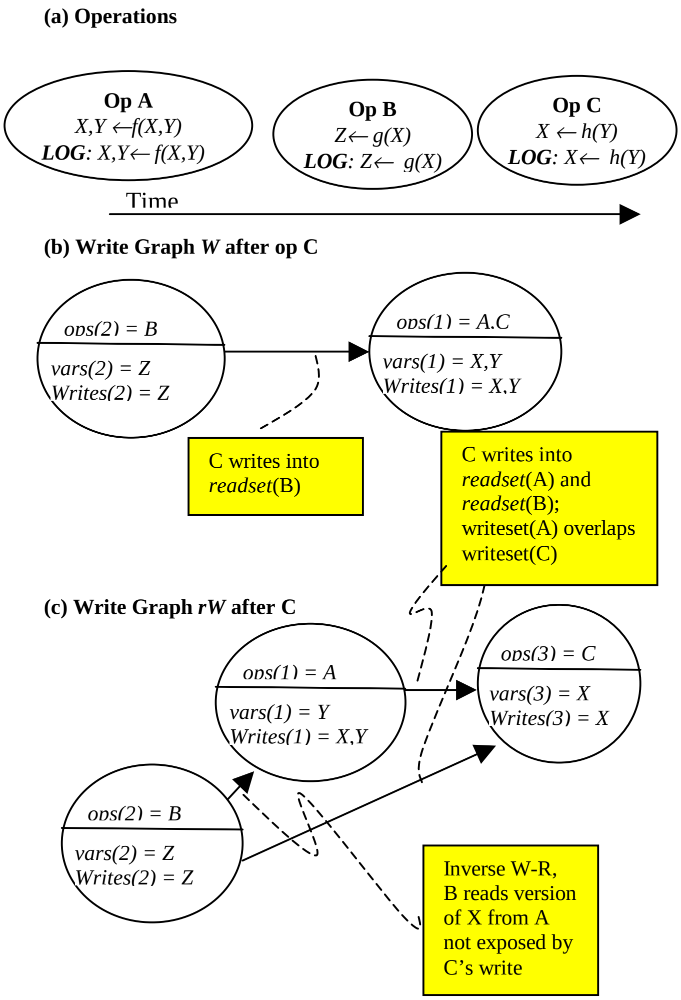

Figure 7: The write graphs *rW* and *W* that result when an object becomes non-exposed by a subsequent operation. *W* has a single node for *X* and *Y* and requires the atomic flushing of both. However, *rW* has separate nodes for *X* and *Y* and the non-exposed object *X* has been removed from *vars(1)*.

Cache Manager Initiated Writes

The cache manager can, via its own actions, cope with *n* having $|vars(n)| > 1$ without needing to atomically flush multiple objects. It **initiates** an identity write $W_{IP}(X)$ on an object *X* in *vars(n).* $W_{IP}(X)$ ‘‘writes" the object without changing it and is logged as a physical operation by writing the value of *X* to the log. This produces a new node *m* with $ops(m) = \{W_{IP}(X)\}$ and $vars(m) = \{X\}$. Importantly, it removes *X* from *vars(n*). No cycles are introduced as *m* will follow other nodes in *rW.* It does not precede any as *Reads(m)* is empty, meaning that $W_{IP}(X)$ has no installation graph successors.

We can repeat this with additional identity writes, until $|vars(n)| = 1$. Once n has no predecessors, the single object can be atomically flushed, installing the operations of *ops(n).* This works for any node *n* with $|vars(n)| > 1$, and hence works with arbitrary log operations. Although only a single object is flushed, all operations of *ops(n)* are installed. (Indeed, we can even reduce $|vars(n)|$ to zero.) Subsequent values for the objects in *Notx(n)*, which are not flushed, can be recovered from the log.

Flushing both installs operations and makes the flushed objects clean (version in the stable state is the same as the cached version). Installation alone does not make stable database and cached versions the same.. Thus, install and flush of an unexposed object leaves that object dirty in cache. The cached version has been updated by an operation that follows the operations being installed. In Figure 7, operation C has updated the cached value of *X* and this value continues to be needed after node *(1)* is installed. The cached value has not been flushed and so is not available from the stable database. Therefore, we continue to require that an object be clean before it can be dropped from the cache to protect our ability to access the latest version of the object, which is needed when subsequent operations read the object.

We have resorted here to logging physical writes to effectively manage the cache when $|vars(n)| > 1$. However, previously, when avoiding flush cycles by precluding logical writes, all writes were physical. Now, we log physical writes only for multi-object atomic flush sets. Even then, we can avoid the need to log at least one object of the set with a physical write. Further, we enable multiple updates to accumulate in each object before we log or flush it. Hence, as is common in database systems, the cost of flushing (and logging) the object is shared among the several updating operations, a substantial saving.

Atomic Flush

It is possible to accomplish atomic flushing of objects in a couple of different ways. So, one might ask, why have we gone to such trouble to try to avoid this. We examine two traditional atomicity techniques here.

1. Shadows: Shadows (used by System R [3]) separate flushing into (i) writing object values to the disk and (ii) including these values in the “official” stable system state (called “propagation” in [5]). When all values are written, one atomically installs them by “swinging” a pointer with a single atomic disk write. With shadows, the entire stable state needs to be shadowed since any part of it might need to be atomically flushed. Shadows relocate objects every time they are written, destroying access sequentiality. Database systems almost universally use logging with update in place, necessitating a different approach to atomic flushing.
2. Flush “transactions”: Database systems achieve atomicity for a set of activities by wrapping them in a transaction. A failure before commit means that none of the activities have “happened”. After commit, all of the activities are guaranteed to have happened. So we can achieve flush atomicity by writing the values of the objects to be flushed all to the log as part of a flush transaction, then writing a commit record for this transaction. Once committed, we can then overwrite the states of the flushed objects in the stable database with the logged values.

To realize a flush transaction, it is important that the states of the objects involved in the transaction be “frozen” in a “flush transaction” consistent state. Hence, we need to protect them from change during the execution of the flush, which includes both logging and subsequently flushing the objects. This kind of consideration is why System R [3] quiesced the system, i.e. paused execution by refusing new actions and completing existing actions, until an action consistent checkpoint was completed.

In terms of I/O costs, each object in the atomic flush set needs to be written twice. The first time, it is written to the log. The log is forced to commit the flush transaction. Then the objects must be updated in place by overwriting them with the values just logged.

Comparing Costs

CM initiated identity writes improve upon flush transactions in two ways.

1. *System interruption is avoided.* There is no requirement to quiesce the system to ensure that “flush transaction” consistent values are written to the stable system state. Rather, we can write values one at a time. Even a subsequent update of an object that is the subject of the CM initiated write can be handled as a normal part of CM operation. So values need be “frozen” only during the time they are actually being written, the same requirement that database systems normally have when managing the cache.
2. *I/O cost is less.* Using CM identity, one object need not be logged prior to being flushed, since single object flushing doesn’t need extra logging. We expect that most multi-object atomic flush sets will be small, mostly of size two, where saving one I/O is important. In this case, we write log two object values when flushing atomically, but only one object value when using CM initiated writes. Further, the normal system operations might remove objects from *vars(n)*, avoiding even more I/O’s. Hot objects will need to be retained in the cache in any event. Hence, we can decide to merely install operations on them via logging, without flushing them immediately, further reducing I/O cost.

An additional benefit to cache manager initiated writes is that we can treat these operations in the same way that we treat regular operations, not as a special mechanism.

5 Recovery **REDO** Tests

To recover the stable database after a crash, the recovery process scans the log sequentially from the redo scan *start point* (log start in [8]) to the end of log. It must determine which operations to replay via the **REDO** test of section 2. **REDO** tests each operation when scanned. If **REDO** returns true, the operation is re-executed using the state *S* formed from the pre-crash stable state as updated by prior redo recovery. We face three related difficulties in to construct an effective **REDO** test for logical log operations.

1. Determining whether an operation is installable when write-write edges between log operations can exist.
2. Avoiding redo of operations in the installed set ***I***, especially when re-execution is costly.
3. Determining if an operation is in ***I*** when it may be installed without flushing its entire *writeset*, as permitted by *rW*.

We want to exploit the explanation of the after-crash state that has the largest set ***L*** of installed operations, and only redo operations in ***H – L***. While determining ***L*** is not always feasible, we desire to use as large an installed set as possible to explain the stable state. Hence, if we can determine that an operation is in ***L***, we want **REDO** to return false and the operation to be bypassed.

SI-based **REDO** Tests

We focus on **REDO** tests based on state identifiers (*SI*'s). (Frequently log sequence numbers (*LSN*s) are used as *SI*'s.) One *SI*, denoted the *vSI*, is stored with each object, and another, the *lSI*, with each log record. For physiological operations, an update of *X* by an operation with a log record whose *lSI* is *k* sets *X*'s *vSI* to *k*. *SI*'s increase monotonically. If *X*'s $vSI \geq lSI$, the operation with $lSI = k$ is in ***L*** and we bypass it. Otherwise, we redo it. This is an effective **REDO** test.

The *SI* based scheme can easily be extended to handle logical log operations when using write graph *W*. We test *SI*'s to determine whether an *Op* is *installed*, not for applicability. We write *vSI*'s for each object in *writeset(Op)* and test objects in *writeset(Op)* to determine whether they contain *Op*'s results. For *W*, the *SI* test returns the same result for all objects in *writeset(Op)* because we atomically install *writeset(Op)*. If $vSI \geq lSI$ for any object in *writeset(Op)*, **REDO** returns false, the operation is in ***L*** and we bypass it. If $vSI < lSI$, *Op* is uninstalled (i.e., it is in ***H - I***) and is re-executed, with objects in *writeset(Op)* set to their *Op* results.

Correct redo recovery requires only that exposed objects have appropriate values. A **REDO** test can return false, meaning redo is unnecessary for operations whose *writeset*'s are entirely unexposed. Consider a system that only logs physical writes $W_P(X,logged(v))$ where *X* is updated from the logged value *v*. We could safely redo all such operations on the log, as they are always applicable and installable. Better would be a **REDO** ``is-installed" test to redo only operations that pass our *SI* test. Even better is to bypass operations on objects until their results are exposed. When all logged operations are blind writes, this means doing nothing until we find the last write for each object. Indeed, media recovery is sometimes performed using a log in which all earlier operations have been deleted [4]. This treats all operations that write to unexposed objects as already installed, regardless of the *SI* test result.

In *rW*, not all of *writeset(Op)* need actually be flushed to install *Op*. So the *SI* test might not return the same result for all objects in *writeset(Op)*. However, because we guarantee atomic installation (not atomic flush), if $vSI \geq lSI$ for any object in *writeset(Op)*, then the operation is manifestly installed. Hence, other objects in *writeset(Op)* with $vSI < lSI$ are established as not exposed.

The traditional *SI* **REDO** test treats all objects with $vSI < lSI$ as if they were exposed. To cope with log operations whose *writeset*'s may be unexposed *and* to exploit the potential substantial gain by treating these operations as installed, we need a **REDO** test for operation *Op* of the following form:

- if an object in *writeset(Op)* has $vSI < lSI$ and is **exposed**, return *true*, i.e., redo *Op*;
- otherwise, return *false*.

Determining whether an object is exposed requires testing more than just its *vSI*, however.

Generalized Recovery *SI*'s

All of *writeset(Op)* may be unexposed. When this occurs, it is possible that $vSI < lSI$ for all variables in *writeset(Op)* but for *Op* to nonetheless be in ***L.*** This is the case trivially in our physical write example above. We want **REDO** to return *false* in this case. While unexposed variables can be set to arbitrary values, and redoing *Op* may not compromise recoverability, we want to avoid the cost of re-execution. This is particularly important for expensive operations like application execution or file writes.

Hence, we introduce the recovery *SI* (*rSI*) for an object (in ARIES, called a recovery *LSN* or *rLSN*). An *rLSN* for a page "indicates from what point in the log there may be updates which are, possibly, not yet in the nonvolatile storage version of the page" [11]. There is an *rSI* for each recoverable object. With physiological operations, which do not have inter-object flush dependencies, the *vSI* of the value of an object that is stored in the stable state indicates the last operation installed for the object. After a cached object is flushed, its *rSI* is reset to the *SI* of the first update after the flush.

LSN's across all objects increase monotonically with each update, not just on a per object basis. ARIES keeps an *rLSN*'s for each dirty object in the CM's dirty pages (dirty objects) table. The minimum *rLSN* identifies the redo scan start point. All operations preceding this minimum *rLSN* are installed. All uninstalled operations are in the tail of the log following this point. This integrates well when *LSN*'s are used because an *LSN* identifies a location in the log. Our **REDO** test can use *LSN*'s as *SI*'s, but requires only that an object's *SI*'s increase monotonically.

ARIES writes to the log the identities of dirty pages and their *rSI*'s in its checkpoint record. Before redo recovery, the latest checkpoint record is retrieved. Its dirty pages are the only pages with uninstalled updates at the time of the checkpoint. Log operations that precede the checkpoint and that involve pages not in the checkpoint record are all installed and hence can be bypassed. This more sophisticated **REDO** test uses the checkpoint *rSI*'s as an adjunct to the *SI* test. However, it only optimizes the *SI* test whose result could always be used, though at the additional cost of reading a page. It does not test if an object is exposed.

We use an *rSI* as part of a **REDO** test that combines an ‘‘is exposed" test with an ‘‘is installed" test. (Operations with unexposed results are “installed”.) An object's *rSI* is the *lSI* of its earliest uninstalled operation (whose results are exposed). The **REDO** test becomes

- if *Op’s* $lSI \geq max(rSI,vSI+1)$ for objects in *writeset(Op)*, *Op* is uninstalled and some result value is exposed. Return *true*;
- otherwise, return *false*.

Below we discuss how to maintain *rSI*’s during normal execution.

We generalized the *rSI* definition to exploit the fact that all operations in *ops(n)* are installed when *vars(n)* are flushed. And this is so even when there are objects in *Notx(n)* that are not flushed. Objects in *Notx(n)* are unexposed because all operations that might have read their values must be installed because of inverse write-read edges. The rule then is that we advance the *rSI* of an object exactly when we install operations that write it, whether or not the object is flushed during installation. Thus, we advance the *rSI*’s of all objects in $Writes(n) = vars(n) \cup Notx(n)$ when *vars(n)* is flushed. An object *X*’s *rSI* is set to the *lSI* of the first **uninstalled** operation to update *X*. This is usually the *lSI* of the first operation with an update that follows the last update of an object in *Notx(n),* as it is typically the subsequent operation’s writing to *X* that make it unexposed. When *X*’s lifetime is terminated, as in a delete, *rSI* becomes the *lSI* of the delete and the object can be removed from the object table. Hence, for a dirty object, $rSI \leq lSI$ of the first uninstalled operation to write *X*, as required in the **REDO** test.

Consider again the example in Figure 7. If node *(1)* is installed via the flushing of *Y*, *X* is also installed and given an updated *rSI* derived from it’s being written by operation A, although *X* was not itself flushed. A couple of points are worth making:

- The *rSI* for *X* is not advanced when operation C is encountered and logged, although prior values of *X* are no longer needed to recover the latest value for *X*. Prior values of *X* may still be needed because other operations read them. In Figure 7, such a value of *X* has been read by operation B.
- The *rSI* for *X* is advanced when node *(1)* is installed because objects in *Notx(1)* are now guaranteed to be not exposed. Hence, their values are not needed by any other operation. *X*’s *rSI* is then set to the *lSI* for operation C.

Logging and Recovery using *rSI*’s

We want to perform our **REDO** test on the log records encountered during the redo recovery scan and only replay operations uninstalled in ***I***, our explaining installed set, i.e. with $lSI \geq max(rSI,vSI+1)$. To use *rSI*’s during recovery requires some action preliminary to performing a redo recovery pass. At a minimum, we retrieve a version of the object table (a dirty object table) that could, as described in ARIES, form a part of our recovery log checkpoint record. Further, a recovery analysis pass preceding the redo pass permits us to exploit the logging of operation installation to generate a dirty object table that reflects the state of dirty objects as of a time close to the crash. Importantly, it lets us remove clean objects from the dirty object table, and to advance *rSI*’s of dirty objects.

The situation is particularly simple for physiological operations. By logging the flush of an object at the point when we know the flush has successfully completed, we are recording not only that the object is now clean but also that prior operations updating the object are installed. During the analysis pass of recovery, when we encounter a “flush” log record, we remove the object from the dirty objects table. If another operation that writes the object is encountered, we return the object to the dirty objects table and set the *rSI* of the object to the *lSI* for this operation. Thus, the *rSI* remains equal to the *lSI* of the first uninstalled operation. This new *rSI* helps during redo recovery with our **REDO** test. Logging object flushes has its origin in recovery lore. Flushes can be lazily logged after the flush as the *vSI* of the object is checked by **REDO** should an update for the object with an *lSI* greater than the *rSI* be present on the log.

In *rW*, flushing objects in *vars(n),* installs operations in *ops(n)*. Flushing is not needed for the unexposed objects of *Notx(n)*. We capture these opportunities to advance object *rSI*’s by logging the installation of each node *n* of *rW*. In that log record, in addition to identifying the objects of *vars(n)* and their *rSI’s,* we identify objects in *Notx(n)* and their *rSI’s*. Recall that the *rSI* for an unexposed object is the *lSI* for the "blind" write (or delete) that follows it. This does for unexposed and exposed objects updated by logical operations what logging of flushes does for physiological operations.

It is possible that *ops(n)* has been installed but the log record describing the installation did not reach the stable log before the system crashed. Thus, we only have approximate information about *rSI*’s during recovery. Sometimes checking *vSI*’s of objects in *writeset(Op),* as with physiological operations, will prevent a needless redo of *Op*. We check the *vSI*’s to ensure that we do not reset objects that are exposed. Operations prevented from being re-executed in this way are manifestly installed. But, all results of *Op* can be unexposed but *Op* can be in ***L*** (the largest set of installed operations explaining state *S*). Unfortunately, **REDO** will indicate that *Op* is uninstalled. Hence, when **REDO** returns true, the operation involved may be:

1) in ***H - I*** for an ***I*** that explains *S*. The operation is then applicable and installable. Re-execution will increase recovery time, but it will not lead to a recovery failure.
2) installed in all ***I*** that explain *S*. The operation may not be applicable as only a minimum uninstalled operation is guaranteed to be applicable. This re-execution may produce erroneous results. If an operation re-execution
    a) only updates the original writeset we do not discover a problem, but no subsequent redo is affected as all objects in *writeset(Op)* are unexposed.
    b) attempts to update more than the original writeset, we can detect this and terminate the redo.
    c) raises an exception when executing against inapplicable state, execution is terminated.

We “expand” **REDO** to include a trial execution of the operation, where if the execution produces errors b) and c), it is "voided". In no case are changes made to exposed objects. Hence, we redo all operations where the *SI* **REDO** test returns true. We can and do re-execute unneeded operations during recovery. Because only the installation(s) just before a crash may be missed, the unneeded extra work will usually be modest.

Recovery optimization using *rSI*'s and logging installations is extremely important when we extend recovery to non-traditional objects such as application state and files. It should usually be possible to avoid both application re-execution and the writing of large files given that both frequently have relatively short lifetimes. Many objects named in log records will, in fact, be terminated or deleted, and so will not be exposed. Hence, one can treat all their operations as installed (i.e. the **REDO** test returns false) even when they have not been flushed recently, or ever.

During recovery, the same conditions on cache flushing apply as during normal operation. Hence, it is necessary to redo operations and cache their results. Only when the re-constructed write graph permits their flushing can operation results be installed. It is possible to avoid reconstructing the write graph by flushing operations as they are completed, exactly as is the case during normal execution. However, it is usually preferable to avoid the frequent I/O and rely on the write graph to guide a more efficient flushing regime.

6 Summary

Our goal has been to promote recovery technology using more general log operations than the now state-of-the-art physiological operations of [4,11]. Logical logging can reduce dramatically the logging required for recovery by using the stable state as a source for values to be exploited during recovery. Physiological logging uses only prior values of the object written by an operation. Logical logging permits many values from the stable state to be used in updating several values per logical operation.

With logical operations, new uses of recovery are possible at reasonable normal execution cost. Our previous paper [7] showed how to provide application recovery when application read operations were handled as logical operations, greatly reducing the cost of logging reads. However, logical operations can result in cyclic flush dependencies, requiring the atomic flushing of multiple objects, a major complication. Write graph *W* is inadequate to avoid this, as once objects need to be flushed together atomically, there is no way to flush them separately.

In [7], we avoided cyclic flush dependencies by restricting ourselves solely to logical reads, logging writes as physical operations, $W_P(X,v)$. This is expensive as the value *v* that updates *X* needs to be logged. The technology introduced in this paper deals with logical operations in general. Hence, it can be used to reduce the cost of handling application write operations by permitting logical write operations $W_L(A,X)$ where the source for *X's* value is the output buffer of the recovered state *A*. Logging $W_L(A,X)$ does not require writing *X*'s new value to the log. This dramatically reduces the normal execution cost of logging application writes.

Because of the subtleties of recovery, we have ranged over a number of subjects.

The installation graph imposes an operation installation order. From it, we derive a write graph that the CM uses to order flushing to keep the system state recoverable. However, the write graph *W* of [8] did not capture the flush requirements with sufficient precision for effective cache management. Therefore, in section 3, we introduced a refined write graph *rW*. *rW* does not require that unexposed objects be flushed in order to install earlier operations that wrote them. Further, the CM, via generation of identity write operations $W_{IP}(X,val(X))$, can separate objects into distinct *rW* nodes, enabling them to be flushed one at a time. As described in section 4, each identity write permits us to remove its updated object from a node's atomic flush set. The identity write puts the object into a node where it could be flushed by itself. Repeating such writes reduces the number of objects in an atomic flush set to only one that can then be flushed. A bonus is that this enables operations to be installed in the stable database without actually flushing some of their updated objects.

Recovery requires a **REDO** test to determine when an operation needs to be re-executed. Traditional *LSN*-based tests require that an operation write a single object and operation installation required flushing the object. We generalized this in section 5 to state identifier (*SI*) based tests that work for arbitrary operations. A new **REDO** test exploited write graph *rW*, where an operation can be installed even when (part of) its *writeset* is not flushed. This required that we use a form of recovery *LSN* called an *rSI* as part of the test.

Recovery for applications or files can be expensive. Since applications and files are frequently only transient objects, executing and disappearing, not recovering them when they are already deleted or not exposed is a significant recovery optimization. Our new **REDO** test, based on *rW* and generalized *rSI*'s, accomplishes that. Using a separate analysis pass, it permits us to identify and redo only operations with exposed updates.

Bibliography

[1] Bernstein, P. Goodman, N. and Hadzilacos, V. *Recovery Algorithms for Database Systems.* IFIP World Computer Congress, (Sept. 83) 799-807.

[2] Crus, R. Data recovery in IBM Database 2. *IBM Systems Journal* 23,2 (1984) 178-188.

[3] Gray, J., McJones, P., et al. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13,2 (June 1981) 223-242.

[4] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann (1993) San Mateo, CA

[5] Haerder, T. and Reuter, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys* 15,4 (Dec. 1983) 287-317.

[6] Kumar, V. and Hsu, M. (eds.) *Recovery Mechanisms in Database Systems*. Prentice Hall, NJ 1998

[7] Lomet, D. Application recovery using generalized redo recovery. *Int'l. Conf. on Data Engineering* , Orlando, FL (February, 1998) 154-163.

[8] Lomet, D. and Tuttle, M. Redo recovery from system crashes. *VLDB Conference*, Zurich, Switzerland (Sept. 1995) 457-468.

[9] Lomet, D. and Tuttle, M. A Formal Treatment of Redo Recovery with Pragmatic Implications. Tech. Report (in preparation).

[10] Lomet, D. Media Recovery When Using Logical Log Operations. (submitted for publication).

[11] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. On Database Systems* 17,1 (Mar. 1992) 94-162.

[12] Strom, R. and Yemini, S. Optimistic Recovery in Distributed Systems. *ACM Trans. On Computer Systems* 3,3 (Aug. 1985) 204-226.