Redo Recovery after System Crashes

David Lomet
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
lomet@microsoft.com

Mark R. Tuttle
Digital Equipment Corporation
One Kendall Square
Cambridge, MA 02139
tuttle@crl.dec.com

Abstract: This paper defines a framework for explaining redo recovery after a system crash. In this framework, an installation graph explains the order in which operations must be installed into the stable database if it is to remain recoverable. This installation graph is a significantly weaker ordering on operations than the conflict graph from concurrency control. We use the installation graph to devise (i) a cache management algorithm for writing data from the volatile cache to the stable database, (ii) the specification of a REDO test used to choose the operations on the log to replay during recovery, and (iii) an idempotent recovery algorithm based on this test; and we prove that these cache management and recovery algorithms are correct. Most pragmatic recovery methods depend on constraining the kinds of operations that can appear in the log, but our framework allows arbitrary logged operations. We use our framework to explain pragmatic methods that constrain the logged operations to reading and writing single pages, and then using this new understanding to relax these constraints. The result is a new class of logged operations having a recovery method with practical advantages over current methods.

1 Introduction

Explaining how to recover from a system crash requires answering some fundamental questions.

- How can the stable state be explained in terms of what operations have been installed and what operations have not?
- How should recovery choose the operations to redo in order to recover an explainable state?
- How should the cache manager install operations into the stable state in order to keep the state explainable, and hence recoverable?

This paper appeared in Umesh Dayal, Peter Gray, and Shojiro Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases,* pages 457-468, September 1995, Morgan Kaufmann Publishers.

The answers to these questions can be found in the delicately-balanced and highly-interdependent decisions an implementor must make. One of these decisions is the choice of operations to be recorded in the log. Many systems rely on page-oriented operations where operations write a single page and read at most that page, but how would the answers to the three questions above change if the operations could read or write other pages? The goal of this work is to understand the impact the choice of logged operations has on crash recovery.

The foundation for this work is an *installation graph* that constrains the order in which changes by operations can be installed into the stable state, and provides a way of explaining what changes have been installed. This graph uses edges to order conflicting operations like the conflict graph from concurrency control, but the installation ordering is much weaker. We prove that if, at the time of a crash, the state can be explained in terms of this graph, then we can use the log to recover the state. We then design a *cache management algorithm* that guarantees that the stable state remains explainable and a *recovery algorithm* that recovers any explainable state, and we prove that these algorithms are correct. Both cache management and recovery algorithms are based on conditions derived from the installation graph. In this sense, it is the installation graph that captures the impact of the choice of logged operations on the recovery process.

One pragmatic impact of our work is a concise explanation of redo recovery. Our work makes it easy to compare recovery methods and to understand how changing the logged operations will change the algorithms needed for crash recovery. For example, given a database using one class of logged operations, we can see exactly how using another class of logged operations forces the cache manager to change. Our work also has the potential for identifying generalizations of known logging strategies. Indeed, we generalize the logged operations used in physiological logging to a class we call *tree operations* that improves the efficiency of logging changes like B-tree splits while introducing only minor changes to cache management.

To the best of our knowledge, this is the first treatment of crash recovery that is both formal and general. Formal treat-

ments of recovery from aborts via transaction rollback are quite general [13], but the only formal treatment of recovery from crashes we know about [8] is specific to ARIES [14]. In the remainder of this introduction, we explain in more detail how the choice of logged operations can affect recovery, and how our framework exposes this impact.

1.1 The basics of redo recovery

When a computer crashes, the contents of the memory are presumed to be lost or corrupted. Only the contents of the disk are trusted to remain. In order to guarantee that the database state can survive a crash, it is stored on disk in the *stable database*. Reading and writing the disk is slow, however, so parts of the state are kept in memory in the *database cache*. Each change an operation makes to the state is made in the cache, and the *cache manager* is responsible for writing these changes into the stable database at some later point in time.

The cache manager may not be able to write all of the changes into the stable database before a crash, so information about each operation is written to a record in a *log*, and the log is used to reconstruct the database state after a crash. The log is typically a sequence of log records (but see [9, 15]). The head of the log is kept on disk in the *stable log*, the tail of the log is kept in memory, and the *log manager* is responsible for moving log records from memory to disk. We assume the log manager is following the *write-ahead log protocol* WAL. This is a well-understood way of managing the log. It has the property that before any state change made by an operation O is written into the stable state, the log record for O and all records preceding this record are written into the stable log.

The recovery process uses the stable state and the stable log to recover the database after a crash. There are many methods for recovery that appear in the literature. These methods provide many specific techniques for ensuring transaction atomicity and durability, including write-ahead logging, the do/undo/redo paradigm, forcing log records at commit, forcing pages at commit, and so on [2, 3, 4, 5, 7, 14], and much of this work has found its way into textbooks [1, 6]. In addition, general methods exist for undoing nested transactions [16] and multi-level transactions [10, 17, 18].

We focus on *redo recovery*. This technique starts at some point in the log and reads to the end of the log. As it examines each log record, it either reinvokes the logged operation on the current state or passes it by. While redo recovery is just one form of recovery, it is an important technique because every efficient recovery method uses it. Redo recovery is efficient in the sense that it does not require the cache manager to write all changes to the stable state when a transaction commits. Redo recovery also has applications in areas like word processing and file editing that are independent of the transactional setting. Furthermore, ARIES [14] suggests understanding recovery after a crash as performing redo recovery followed by undo recovery. With this, redo recovery must solve the hardest parts of recovery: making sense of the state at the time of the crash, and determining what operations in the log to reexecute to rebuild the state. These are the kinds of problems we want to study.

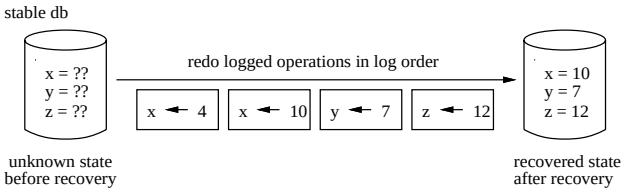

Figure 1: Recovery based on after-image logging.

1.2 The problem

The operations a user invokes on the database can be quite different from the operations appearing in the log. For example, inserting a record into a B-tree may be a single operation to the user, but may be recorded as a sequence of write operations in the log if the insert involves splitting a node. The only requirement is that the combined effect of the logged operations be equal to the effect of the user operation. There are many ways that a user operation can be split into logged operations. How this split is made can affect many aspects of recovery. To see that this is true, let us examine two common logging strategies.

In the simplest form of *after-image logging* (also called *state* or *physical logging*), each log record consists of a variable name and a value written to that variable. Each logged operation is effectively of the form $x \leftarrow v$ describing a blind write to a single variable (typically a disk page). This technique has the advantage that cache management and recovery are quite easy. The cache manager can install changes into the stable database in any order, and recovery is still possible. In the example in Figure 1, variables in the stable database can have any value at the start of recovery. The recovery process merely needs to start scanning the log at a point where all uninstalled operations will be included in the scan, and then write values into variables one after another in log (or conflict graph) order. At the end of recovery, each variable will be set to its final logged value and the state will be recovered.

In *logical logging*, each log record consists of the name of the operation and the parameters with which it was invoked. This technique has the advantage that the log records are much smaller (an operation's name and parameters are typically much smaller than a copy of a disk page), but now cache management is tricky. In the scenario of Figure 2, the stable database and the database cache begin in the same state, then two swap operations are performed, their changes are made in the cache, and the two log records for the swaps

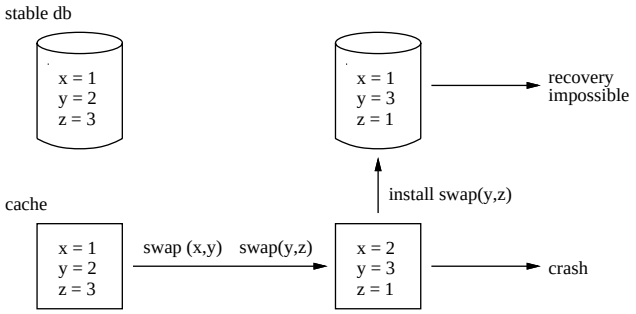

Figure 2: Recovery based on logical logging.

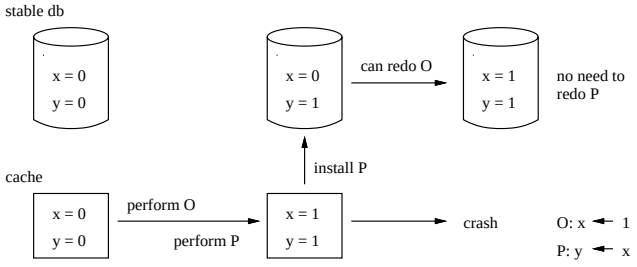

Figure 3: Write-read edges can be ignored.

are written to the stable log. Suppose that the effect of the first swap is written to the stable database just before a crash. The recovery process can redo the second swap and recover the state of the database. On the other hand, suppose the effect of the second swap is installed just before a crash, and the effect of the first is never installed. Now the recovery process is stuck. There is no way that it can recover by redoing one or both of the swap operations appearing in the log: it can't swap the value 2 into x since the value 2 does not even appear in the stable state. The cache manager *must* install the first swap before the second.

Given a class of logged operations, how much flexibility does the cache manager have in the order it installs database changes? At one extreme, the cache manager could install changes in the order they occur, but this ordering is too strict to be efficient. At the other extreme, the cache manager could install changes in any order. This does not work with logical logging. On the other hand, it does work with after-image logging. This causes us to ask the following question: given a set of logged operations, what is the weakest possible ordering that the cache manager can use when installing changes and still guarantee that the stable database remains recoverable?

Identifying this weakest ordering is interesting for many reasons. It gives the cache manager more flexibility to install changes in a more efficient order, perhaps making cache management simpler. It also may lead to new logging strategies making use of new kinds of logged operations that reduce logging cost but preserve cache efficiency and database recoverability. Finally, it can lead to improved understanding and comparison of known recovery strategies.

1.3 The solution

Given a class of logged operations, we represent the amount of flexibility the cache manager has to install the changes made by these operations with an *installation graph* giving a partial order on the operations. We prove that if a database state can be "explained" as the result of installing operations in a prefix of this graph, and if the database log contains the remaining uninstalled operations, then the database is recoverable. One well-known partial order on operations is the *conflict graph*. Two operations O and P conflict if one writes a variable x that the other reads or writes, and every pair of conflicting operations is ordered by an edge in the graph from one to the other. The edges are classified into sets of read-write, write-read, and write-write edges depending on how the operations O and P access x. It is easy to see that if operations are installed in a sequential order consistent with the conflict graph, then the state remains recoverable: redoing the uninstalled operations in a sequential order consistent with the graph will recover the state. Are all of the edges in the conflict graph needed? The answer is no.

We do not need write-read edges. The example in Figure 3 involves one operation O setting $x \leftarrow 1$ followed by a second operation P setting $y \leftarrow x$, and there is clearly a write-read edge from O to P in the conflict graph since O writes x before P reads it. O and P are invoked on the database and their changes are made in the cache, but P is installed first in the stable database just before a crash. The recovery process can redo O, and since redoing O does not reset the effect of P (O does not write to the variable y written by P), recovery is accomplished.

In many cases, we do not need write-write edges either. This observation is related to the Thomas write-rule in the special case of blind writes to single pages, but it is also true in more general situations. In the example of Figure 4, an operation O setting $x \leftarrow 1$ and $z \leftarrow 2$ is followed by an operation P setting $y \leftarrow x$ and $z \leftarrow 3$, so there is a write-write edge from O to P since both write to the variable z. Again, O and P are invoked on the database and their changes are made in the cache, but only P is installed in the stable database before a crash, and not O. The recovery process can redo O and this resets a change made by P, but redoing O has the effect of reconstructing the read set of P, so the recovery process can redo P as well.

We can generalize this last example to the case where redoing a sequence of operations beginning with O reconstructs the read set of P during recovery, and hence enables the redo of P. Given an operation O, we define the *must redo set of* O, denoted by $must(O)$, that contains O and all operations following O in the conflict graph whose changes would

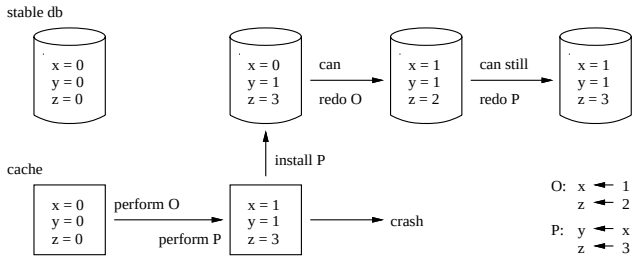

Figure 4: Write-write edges can often be ignored.

be reset by redoing O during recovery, and hence would have to be redone themselves. This is effectively the transitive closure of the write-write edges in the conflict graph starting from O. We also define the *can redo set of* O, denoted by $can(O)$, containing operations reset by redoing O that can ultimately be redone as a result of redoing O. These operations can be ordered in a way consistent with the conflict graph and beginning with O so that redoing the first operation rebuilds the read set of the second, redoing the first two operations rebuilds the read set of the third, and so on.

The *installation graph* is obtained from the conflict graph by keeping all read-write edges, throwing away all write-read edges, and keeping write-write edges from O to P when $P \in must(O) \backslash can(O)$.[1] This reflects our earlier observation that all of the write-read edges and many of the write-write edges are unnecessary. In the case of write-write edges, there is an edge from O to P if redoing O during recovery forces us to redo P, but redoing O does not guarantee that P can actually be redone. This definition of an installation graph is quite simple, but it is already enough to demonstrate differences between logging strategies.

First consider after-image logging, where the logged operations are blind writes of the form $x \leftarrow v$. In this case, the installation graph has no edges at all, which corresponds to our earlier observation that changes resulting from these operations can be installed in any order. This is because the read set of every operation is empty. There are obviously no read-write edges for this reason. To see that there are no write-write edges, notice that if P is in O's must redo set then P is in O's can redo set, since P's read set is null and does not need reconstruction.

Now consider the case of physiological logging [6], where each logged operation $x \leftarrow f(x)$ reads the value of a single page and writes some function of this value back to the same page. In this case, the installation graph consists of chains of read-write edges, one chain for each variable x that orders the operations accessing x. Once again there are no write-write edges because every operation P in O's must redo set is also in O's can redo set. To see this, notice that if O reads and writes x, then O's must redo set consists of all operations P reading and writing x that follow O in the conflict graph. We can redo all of these operations after redoing O by redoing them in conflict graph order.

The installation graph we have defined so far is quite simple because we have made some simplifying assumptions. We have assumed that when an operation O's changes are installed into the stable database, that all of its changes are installed atomically, but this is frequently not necessary. Atomic installation may even be problematic if an operation O can write multiple pages. We allow O's write set to be partitioned into *updates* that represent the units of atomic installation. We have also assumed that when an operation O is redone during recovery, that the entire write set of O is recomputed and installed, but this may not be true either. There are recovery methods that can test during recovery which of O's changes are uninstalled and recompute only a portion of O's write set. One example is recovery based on log sequence numbers (LSN's), where every page is tagged with the LSN of the log record of the last operation to write to the page. During recovery, any page having an LSN greater than the LSN associated with an operation's log record must already contain the changes made by this operation, so no change needs to be made to the page. We allow O's write set to be partitioned into *redos* that represent the units of recovery. Incorporating these notions into the installation graph is delicate, but we do so in a clean way, and this enables us to capture more general logging strategies.

1.4 The consequences

Given our installation graph, we prove that if the state of the stable database can be "explained" by a prefix of the installation graph, then we can use the stable log to recover the state. A state S can be explained by a prefix of the graph if there is an ordering of the operations in the prefix that (among other things) assigns certain "exposed" variables the same values they have in S. If we consider the operations in the prefix to be installed and the rest of the operations to be uninstalled, then the exposed variables are (roughly speaking) all variables that are read by one uninstalled operation before being written by any other. Assuming these exposed variables have the right values, then we can recover the state simply by redoing the uninstalled operations in conflict graph order.

Since an explainable state is a recoverable state, all the cache manager has to do is install operations one after another in a way that guarantees that the stable state remains explainable. Of course, in reality, a cache manager does not install operations, it installs variables. If two operations happen to write two values to a variable x between two successive installations of x, then the cache manager writes the second value back to the stable state and effectively installs changes made by both operations at once. Nonetheless, the installation graph ordering of operations can be used to construct an ordering on variables so that if the cache manager installs variables in this order, then the state remains explain-

[1] We denote the set difference of two sets A and B by $A \backslash B$. This is the set of elements in A and not in B.

able in terms of a set of installed operations.

Recovering an explainable state can be as simple as going through the log and redoing an operation if it is uninstalled, but this requires a test for whether an operation is installed. On the other hand, some recovery methods redo many more operations than this. Some recovery methods go through the log and redo every operation that can be redone, but this requires knowing whether an operation is applicable in the current state. This also requires knowing whether redoing an installed operation O will reset the effects of another installed operation P, and if so, whether it will be possible to redo P when it is reached in the log. We abstract away these details by assuming the existence of a test $\text{REDO}(O)$ for whether the recovery procedure should redo the operation O. The specification of $\text{REDO}(O)$ is stated in terms of the installation graph, and it is an abstraction of many implementation techniques like log sequence numbers used to determine during recovery whether the operation O should be redone. We give an algorithm that goes through the log and redoes O whenever $\text{REDO}(O)$ returns true, and we prove that this is an idempotent recovery algorithm when started in an explainable (and hence recoverable) state.

The installation graph is also a criterion for the correctness of checkpointing algorithms. The process of checkpointing need only identify (perhaps after flushing portions of the cache to disk) a prefix of the installation graph that explains the stable state, and then remove a prefix of the stable log that contains only operations in this prefix, always leaving all uninstalled operations on the log.

Finally, we can exploit the installation graph to generalize existing logging strategies. We show how the logged operations used in physiological logging can be generalized to what we call *tree operations*. A tree operation reads one page and writes that page together with a set of "new pages" that have never been written before. Cache management for tree operations is only slightly more difficult that for physiological operations. Tree operations, however, enable us to log the splitting of a B-tree node with a single operation rather than with a sequence of physiological operations. In addition, we can use checkpointing as a kind of garbage collection that allows us to reclaim used-and-discarded pages as "new pages" that can be written by future tree operations. In ways like these, the installation graph gives us a graph-theoretic technique for understanding many aspects of redo recovery methods that appear in the literature.

2 Database Model

We sketch our model of a database here, and give the complete statement in the technical report [12]. A *state* is a function mapping variables in a set V to values in a set $\mathcal{V}$, and one state is chosen as the *initial state*. An *operation* is a function mapping states to states. We consider only the logged operations in this model, and do not explicitly model the user operations. An *operation invocation* describes a particular invocation of an operation. It is a tuple including the operation invoked, the read and write sets for the invocation, and their before and after images. When it will cause no confusion, we denote an operation invocation by the name of the operation invoked, and we shorten "operation invocation" to "operation."

We mentioned in the introduction that the changes an operation makes to its write set need not be installed into the database atomically, since they may be installed as a sequence of atomic installations instead, perhaps one page at a time. We model this by assuming that the write set of an operation O is partitioned into a collection $U_1, \ldots, U_u$ of disjoint subsets. Our interpretation of U_i is that the changes O makes to variables in U_i must be installed atomically. We refer to the pair $\langle O, U_i \rangle$ as an *update*. We assume that the intersection of O's read and write sets is contained in just one of the sets U_i, which means that O's changes to its read set must be installed atomically. We note that O's changes to its read set must be installed after all of O's other changes, or it may be impossible to redo O after a crash. Informally, we think of an operation O as being a set of updates whose write sets partition O's write set.

We also mentioned in the introduction that it may not be necessary or desirable to reconstruct the entire write set of an operation during recovery, since it may be possible to apply some test during recovery to determine which of the operation's updates need to be reinstalled. The more specific this test can be about which updates need to be reinstalled, the smaller the reconstructed portion of the write set needs to be. We model this by assuming that an operation O's set of updates is partitioned into a collection of *redo's*. The recovery process may choose to redo only a few of an operation's redos, but redoing one redo means reinstalling all of that redo's updates. During recovery, a redo is an all or nothing thing: either all of the updates in the redo are reinstalled, or none are. Given an update $\langle O, U \rangle$, we let $[O, U]$ denote the redo that contains it.

Two operation invocations *conflict* if the write set of one intersects the read or write set of the other. A *conflict graph* orders conflicting invocations O and P with a path from O to P if O occurs before P, in which case we write $O \leq P$. We consider only *serializable* conflict graphs, which means they can be totally ordered. A *history* H is a conflict graph describing a database execution; that is, a conflict graph that contains all operation ever invoked on the database. A *log* L for H is a subgraph of H induced by some nodes in H giving a partial description of the history, and possibly including some additional edges to impose a stricter ordering (like a sequence) on invocations.

A *database system* D is a triple (S, L, H) consisting of a state S, a history H, and a log L for H. Of course, no real database implementation would explicitly represent the history since the state and the log contain all of the information about the history needed by the implementation. For us,

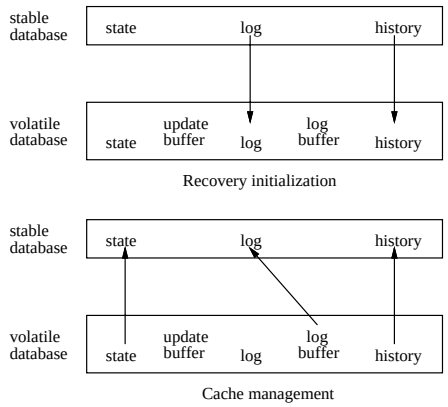

Figure 5: Normal operation and recovery in a database.

the history is just the correctness condition for the recovery process. We say that D is *recovered* if its state S is the result of starting with the initial state and applying the operations in its history H in a sequence consistent with H. We say that D is *recoverable* if it can be transformed into a recovered database by installing some redos for some operations appearing in the log in some order consistent with the log. The recovery process should be *idempotent*, meaning that the database remains recoverable throughout the recovery process.

We can assume that the log is always a suffix of the history, for two reasons. First, we assume that the log manager is following the *write-ahead log protocol* WAL. In our model, this means that when an operation is added to the stable history, the operation is also added to the stable log. Second, we assume that the *checkpointing* process—which speeds recovery by shortening the log—removes installed operations from the front of the log. The result is that the log forms a suffix of the history that contains all uninstalled operations of the history. For example, this means the log of a recovered database can be empty.

A database implementation splits a database into stable and volatile components as illustrated in Figure 5. The *stable database* is a database (S_s, L_s, H_s) as defined above, and represents the disk resident part of the database. The *volatile database* or *cache* is a partial database $(S_v, L_v, H_v, U_v, B_v)$—meaning that the volatile state is a partial mapping from variables to values—with two additional data structures:

- the *update buffer* U_v is a set of updates not yet installed in the stable state S_s, and
- the *log buffer* B_v is a working log consisting of operations not yet posted to the stable log L_s.

The logical database is a database (S_ℓ, L_ℓ, H_ℓ) where the state S_ℓ is a merge of the volatile and stable states, the log L_ℓ is the volatile log, and the history H_ℓ is the volatile history. The value of a variable x in the logical state is the value of x in the volatile state if defined, and the value of x in the stable state otherwise.

During normal operation, invoking an operation O on the database changes the volatile state by changing the values of variables in O's write set to the values given by O's after image, adds O's updates to the update buffer, adds O to the log buffer, and adds O to the volatile history. The volatile log is empty during normal operation—it is used only during recovery—implying that the logical log is also empty.

At the start of recovery, the volatile database is reset as follows (see Figure 5). The volatile log and history are set equal to the stable log and history; and the volatile state, update buffer, and log buffer are set to empty. One after another, operations O are chosen from the front of the log, and some collection of O's redos $[O, U]$, perhaps none, are chosen. For each chosen redo $[O, U]$, one after another, the updates $\langle O, V \rangle$ in $[O, U]$ are used to modify the volatile state and then are added to the update buffer. Finally, O is removed from the log. Notice that O already appears in the stable history and log, so there is no need to add it to the volatile history and log buffer.

It is the cache manager that moves information from the volatile database to the stable database (see Figure 5). It does so as follows:

1. It chooses a variable x to write to the stable state, and then chooses a set X of variables including x. How this choice is made is the subject of a later section.
2. It updates the stable log by following the write-ahead log protocol. This means that it identifies all operations O with updates $\langle O, U \rangle$ in the volatile update buffer writing to a variable in X, it identifies prefixes of the volatile log buffer and volatile history so that no such operation O appears in the remaining suffixes, it adds these prefixes to the stable log and stable history, and it removes the log buffer prefix from the log buffer.
3. It writes X atomically to the stable state and removes the updates $\langle O, U \rangle$ from the volatile update buffer.

This description is consistent with our use of the WAL protocol and our checkpointing requirement, and guarantees that the stable log is always a suffix of the stable history.

3 Conditions for Recoverability

The main results of this section are the definition of the installation graph and what it means for a state to be explainable in terms of this graph. Then in later sections we will use this graph to construct recovery and cache management algorithms.

3.1 Must redo and Can redo

As in the introduction, the definition of the installation graph begins with the definition of the must and can redo sets.

In the introduction, we defined $must(O)$ to be the set of all operations following O in the conflict graph whose changes would be reset by redoing O during recovery, and hence would have to be redone themselves. This was effectively the transitive closure of the write-write edges in the conflict graph starting with O. Now that redos are the unit of recovery, and not operations, we want to define this set in terms of updates and redos.

Assuming that we must reinstall an update $\langle O, U\rangle$ during recovery, what is the set of updates that will be deinstalled when we reinstall $\langle O, U\rangle$? First, reinstalling $\langle O, U\rangle$ during recovery requires that we reinstall every update in its redo set $[O, U]$. Next, reinstalling the updates in $[O, U]$ may overwrite variables written by operations following O in the conflict graph, effectively deinstalling updates for later operations. Each of these deinstalled updates $\langle P, V\rangle$ and their associated redos $[P, V]$ will have to be reinstalled, and reinstalling them will in turn deinstall updates for later operations, and so on.

With this intuition, we define $|O, P|$ to be the length of the longest path from O to P in the conflict graph, and we define $\text{MUST}\langle O, U\rangle$ by induction as follows.

1. $\text{MUST}_0\langle O, U\rangle = [O, U]$.
2. $\text{MUST}_d\langle O, U\rangle = \text{MUST}_{d-1}\langle O, U\rangle \cup [P, V]$ for all updates $\langle P, V\rangle$ such that $d = |O, P|$ and V intersects the set of variables written by updates in $\text{MUST}_{d-1}\langle O, U\rangle$.

The set $\text{MUST}\langle O, U\rangle$ is the union or limit of the sets $\text{MUST}_d\langle O, U\rangle$, and $\text{MUST}\langle O, U\rangle \backslash [O, U]$[2] is the set of updates that must be reinstalled as a result of reinstalling $\langle O, U\rangle$. Notice how the size of the redos $[P, V]$ affects the size of the must redo sets $\text{MUST}\langle O, U\rangle$: the more updates in a redo $[P, V]$, the more updates that have to be reinstalled after $\langle O, U\rangle$.

In the introduction, we defined $can(O)$ to be the operations in $must(O)$ that we know can ultimately be redone once O has been redone. These operations can be ordered in a way consistent with the conflict graph and beginning with O so that redoing the first operation rebuilds the read set of the second, redoing the first two operations rebuilds the read set of the third, and so on. This is because the second operation reads the variables just written by the first one, the third operation reads the variables just written by the first two, and so on. Since each operation will read the same values from its read set during recovery as it saw during normal operation, it will write the same values to its write set. We can make a similar definition in terms of updates and redos. We proceed by induction and define

1. $\text{CAN}_0\langle O, U\rangle = [O, U]$.
2. $\text{CAN}_d\langle O, U\rangle = \text{CAN}_{d-1}\langle O, U\rangle \cup [P, V]$ for all updates $\langle P, V\rangle$ in the set $\text{MUST}_d\langle O, U\rangle$ with $d = |O, P|$ such that the read set of P is contained in the set of variables written by updates in $\text{CAN}_{d-1}\langle O, U\rangle$.

The set $\text{CAN}\langle O, U\rangle$ is the union or limit of the $\text{CAN}_d\langle O, U\rangle$.

Having thought so hard about how installing one update during recovery can deinstall a second, let us formally define an installed update. Suppose we have a state that is the result of installing some sequence σ of updates, where σ may describe alternating periods of normal operation and recovery. Which of the updates in σ do we consider installed? We have already noted that $\text{MUST}\langle O, U\rangle \backslash [O, U]$ is the set of updates that must be redone as a result of reinstalling $\langle O, U\rangle$, so it seems natural to define *installed subsequence* of σ to be the subsequence of σ obtained as follows:

1. Select an update $\langle O, U\rangle$ in σ and delete from σ all but the last instance of $\langle O, U\rangle$ in σ.
2. Delete from σ all instances of updates in $\text{MUST}\langle O, U\rangle \backslash [O, U]$ that precede this last appearance of $\langle O, U\rangle$ in σ,
3. repeat this for all updates in σ.

The result of this construction is well-defined; that is, if σ_1 and σ_2 are installed subsequences of σ, then $\sigma_1 = \sigma_2$. We define the set of *installed updates* in σ to be those appearing in the installed subsequence. A sequence σ is an *installed sequence* if it is equal to its own installed subsequence.

3.2 Installation Graph

In the introduction, we defined the installation graph by observing that, in the conflict graph, all of the write-read edges and many of the write-write edges are unneeded. We defined the installation graph to be the result of keeping all read-write edges, throwing away all write-read edges, and keeping write-write edges from O to P when $P \in must(O) \backslash can(O)$. Keeping the write-write edge from O to P meant that redoing O meant we must redo P, but that redoing O is no guarantee that we can ever redo P. Since must and can redo sets have exactly the same interpretation when defined in terms of updates and redos, we can define the installation graph in exactly the same way now.

The *installation graph* for a history (or conflict graph) is a directed graph where each node is labeled with an update $\langle O, U\rangle$, and for distinct updates $\langle O, U\rangle$ and $\langle P, V\rangle$, there is an edge from $\langle O, U\rangle$ to $\langle P, V\rangle$ if $O \leq P$ in the history and either

1. *read-write edges*: the intersection of the read set of O and V is nonempty, or
2. *write-write edges*: $\langle P, V\rangle$ is contained in $\text{MUST}\langle O, U\rangle$ but not in $\text{CAN}\langle O, U\rangle$.

We write $\langle O, U\rangle \prec \langle P, V\rangle$ if there is a path from $\langle O, U\rangle$ to $\langle P, V\rangle$. A *prefix* of the installation graph is an update sequence σ such that (i) σ is an installed sequence, and (ii) if

[2] Remember that we write $A \backslash B$ to denote the set of elements in A that are not in B.

$\langle O, U \rangle$ appears in σ, then every update $\langle P, V \rangle \prec \langle O, U \rangle$ appears in σ. We will also refer to a set I of updates as a prefix if there is a prefix σ consisting of the updates in I.

The updates for an operation O are unordered by $\prec$ for the most part, the only exception being when O writes to its own read set. In this case, there is a read-write edge from every update $\langle O, V \rangle$ to the (single) update $\langle O, U \rangle$ writing to O's read set. These read-write edges represent the fact that O's changes to its read set must be installed last, or it may be impossible to redo O after a crash. We say that $\langle O, U \rangle$ is the *final* update for O if U contains O's read set, and we say that $[O, U]$ is the *final* redo for O if $[O, U]$ contains O's final update. A *consistent* ordering of O's updates is any sequence that ends with O's final update or does not include O's final update, and we define a *consistent* sequence of O's redos in the same way. Ordering updates in this way has been referred to as "careful replacement" in the recovery literature [1, 6].

Another useful ordering on updates respects both the installation graph and conflict graph orderings: we define $\langle O, U \rangle \sqsubset \langle P, V \rangle$ iff $\langle O, U \rangle \prec \langle P, V \rangle$ or $O < P$. It is convenient to extend $\sqsubset$ to redos: we define $[O, U] \sqsubset [P, V]$ iff $[O, U]$ and $[P, V]$ contain $\langle O, U' \rangle$ and $\langle P, V' \rangle$ satisfying $\langle O, U' \rangle \sqsubset \langle P, V' \rangle$. When we say that an update $\langle O, U \rangle$ or redo $[O, U]$ is minimal with respect to some set of updates or redos, we mean minimal with respect to $\sqsubset$. The idea here is that we will be redoing operations in conflict order during recovery because we read the log in this sequence. We install the updates of these operations in an order consistent with the installation graph.

3.3 Explainable States

During recovery, certain important variables must have the right values in them. These are the variables whose values must be correct in order to reconstruct the read sets of the uninstalled updates. We define a variable x to be *exposed* by σ iff one of the following conditions is true:

1. no update uninstalled after σ reads or writes x, or
2. some update uninstalled after σ reads or writes x, and the minimal such update reads x.

The exposed variables have some very nice properties. For example, if σ and τ have the same set of installed updates, then they have the same set of exposed variables, and every exposed variable x has the same value after σ and τ.

A prefix σ *explains* a state S if for every variable x exposed by σ, the value of x in S is the value of x after σ. A set of installed updates I *explains* a state S if some σ explains S and I is the set of installed updates in σ, in which case we can prove that any τ with this set of installed updates explains S. Thus, if S is the state of the stable database at the start of recovery, then the exact sequence of updates used to construct S is unimportant. Only the set of updates 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, a task that is made even more difficult by the fact that crashes can occur that require multiple invocations of recovery and hence produce arbitrarily complex installation sequences.

3.4 Minimal Uninstalled Updates

An operation O is *applicable* to a state S if for every variable x in O's read set, 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. An update $\langle O, U \rangle$ is *installable* in a state S if the database state $S' = S\langle O, U \rangle$ obtained by installing the effects of $\langle O, U \rangle$ into S is explainable by a prefix of the installation graph. We can extend this definition to sequences of updates in the obvious way.

A prefix σ can be *extended* by an update $\langle O, U \rangle$ if (i) there is no write-write edge from $\langle O, U \rangle$ to any update in σ, and (ii) σ contains every update $\langle P, V \rangle \prec \langle O, U \rangle$. We define *extend*$(\sigma, \langle O, U \rangle)$ to be the result of deleting every update in $\text{MUST}\langle O, U \rangle \backslash [O, U]$ from σ, and then appending $\langle O, U \rangle$ if it does not appear in the result. This is the result of removing all updates that $\langle O, U \rangle$ deinstalls, and then making sure that $\langle O, U \rangle$'s effects are present.

The following theorem is the basis of our informal intuition that an explainable state can be recovered by installing uninstalled updates in conflict graph order.

Theorem 1 *Let S be an explainable state. If $[O, U]$ is a minimal redo with an uninstalled update and τ is any consistent ordering of $[O, U]$, then O is applicable to S and* extend(σ, τ) *explains $S\tau$, so τ is installable in S.*

4 General Recovery Method

Theorem 1 suggests a procedure for recovering a database with an explainable state: choose a minimal redo $[O, U]$ with an uninstalled update, install it, and repeat. In order for this to work, of course, O must appear in the log. Informally, we say that a database is explainable if its state is explainable and its log contains the uninstalled operations. Formally, we say that a database $D = (S, L, H)$ is *explainable* if there is a prefix σ of the installation graph of the history H such that σ explains the state S and σ contains every update of every operation that is in the history H but not in the log L.

The procedure Recover(D, σ) in Figure 6 captures the procedure described above for recovering a database D explained by a prefix σ. The algorithm considers all operations O in log order, and considers all redos for O in a consistent order, and then invokes a test $\text{REDO}(D, \sigma, [O, U])$ to determine whether $[O, U]$ should be installed into the state of a database D explained by a prefix σ.

But under what conditions should this test $\text{REDO}(D, \sigma, [O, U])$ return true? At the very least, it

```
procedure Recover(D, σ)
    while the log L is nonempty do
        choose a minimal operation O in the log L
        choose a consistent ordering of O's redos
        for each redo [O, U] satisfying REDO(D, σ, [O, U])
          do
            compute the after image of [O, U]
            choose a consistent ordering of [O, U]'s updates
            for each update ⟨O, V⟩ do atomically
                install ⟨O, V⟩
                replace σ with extend(σ, ⟨O, V⟩)
        delete O from the log L
```

Figure 6: Recovering a database D explained by σ.

must return true if $[O, U]$ contains an uninstalled update. Since the log is read in conflict graph order, such a $[O, U]$ would be a minimal redo with an uninstalled update, and in this case Theorem 1 says that $[O, U]$ is guaranteed to be applicable and installable. Sometimes it is hard to tell that a redo contains an uninstalled update, so a recovery method may end up redoing many installed redos all over again. In fact, whenever $[O, U]$ is applicable and installable, it is okay for the test to return true. Since the log is read in conflict graph order, we know that installing $[O, U]$ is not going to deinstall things the recovery process has just installed, and deinstalled things will be reinstalled when we reach them later in the log.

With this in mind, we require that the test $\text{REDO}(D, \sigma, [O, U])$ satisfies the following conditions:

1. Safety: If the test returns true, then (a) O is applicable to S, and (b) there are no write-write edges from updates in $[O, U]$ to updates in σ
2. Liveness: If $[O, U]$ is a minimal redo containing an uninstalled update in σ, then the test returns true.

Theorem 1 says that the liveness condition implies the safety condition, and the safety condition essentially says that $[O, U]$ must be applicable and installable: since operations are consider in conflict graph order, condition (1b) will imply that $[O, U]$ extends σ and hence is installable. This test need only be defined for a database D that is explainable by a prefix σ and for a redo $[O, U]$ where O is a minimal operation on D's log, since these are the only conditions under which the test is used. The safety condition guarantees that an operation is redone only when it sees the read set seen originally and when installing its redo's will reset the state such that operations that are deinstalled by it can be redone. The liveness condition guarantees that the uninstalled operations will always be redone and reinstalled during recovery.

One property of the algorithm Recover that will bother some people is the fact that it maintains a sequence σ of updates. No recovery algorithm actually does this. Fortunately, σ is only used when evaluating the test $\text{REDO}(D, \sigma, [O, U])$. Since testing for applicability is usually much easier that testing for installability, most practical methods restrict their operations so that applicable operations are always installable. For example, page-oriented and tree-structured operations that we study later in this paper restrict their operations so that there are effectively no write-write edges in the installation graph. Thus, $\text{REDO}(D, \sigma, [O, U])$ can ignore σ and return true whenever O is applicable. (This becomes a particularly easy test for blind writes as they are always applicable. For page-oriented read/write operations, we can exploit state identifiers to determine when a page is in the state in which the operation was originally done.) Since σ is unneeded by the test, maintenance of σ can be removed from the algorithm. We can prove that the invariant "D is explained by σ" holds after each step of Recover(D, σ), and conclude that an explainable database is recoverable.

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

5 General Cache Management

The main result of the previous section was that an explainable database is a recoverable database. To prove that crash recovery is possible, we need to show how to keep the stable database explainable; that is, so that the stable state is explainable and the stable log contains all of the uninstalled operations. Fortunately, the WAL protocol and our checkpointing requirement guarantee that the all uninstalled operations appear in the stable log. Specifically, the WAL protocol guarantees that an operation is added to L_s the moment it is added to H_s, and our checkpointing requirement guarantees that only installed operations are removed from L_s, so together they guarantee that operations in $H_s \backslash L_s$ are installed. Consequently, in this section, we just need to show how to keep the stable state S_s explainable. We give an algorithm for managing the cache during normal operation and recovery that keeps the stable state in an explainable state.

We have assumed that the volatile update buffer is a set of updates whose effects appear in the volatile state. Here, we must assume that the update buffer is actually the subgraph of the installation graph induced by the updates in the update buffer. We discuss how to avoid explicitly maintaining this subgraph in the technical report [12].

A cache manager effectively partitions the volatile state into a "dirty" part and a "clean" part (which we do not discuss here). A variable 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 only and immediately upon being written to the stable database. Variables of the dirty volatile state are written to the stable database for two reasons. First, the volatile state can be

procedure WriteGraph(I)
  $T \leftarrow$ the transitive closure of "$\langle O, U\rangle \sim_I \langle P, V\rangle$ iff $U \cap V$ is nonempty" for nodes of I
  $\mathcal{I} \leftarrow$ the graph I after replacing each $\langle O, U\rangle$ with $\{\langle O, U\rangle\}$
  $\mathcal{V} \leftarrow$ collapse $\mathcal{I}$ with respect to the equivalence classes of T
  $S \leftarrow$ the strongly connected components of $\mathcal{V}$
  $\mathcal{W} \leftarrow$ collapse $\mathcal{V}$ with respect to the equivalence classes of nodes in S
  return($\mathcal{W}$) /* collapsing $\mathcal{V}$ made $\mathcal{W}$ acyclic */

Figure 7: Computing the write graph.

(nearly) full, requiring that variables currently present be removed to make room for new variables. Second, it may be desired to shorten recovery by checkpointing the stable log. Since only installed operations can be removed from the log, it may be necessary to install some of their updates before removing them from the log. Systematic installation permits a prefix of the log to be truncated while preserving stable database recoverability.

The central problem for cache management is that installation graph nodes are *updates* but the cache manager writes *variables*. The cache manager must write sets of variables in such a way that update atomicity and update installation order are observed. The cache manager computes a *write graph* for this purpose. Each write graph node v has an associated set $updates(v)$ of updates in the volatile update buffer and a set $variables(v)$ of the variables these updates write. The variables of a write graph node must be written atomically in order to guarantee update atomicity. These sets must be written in write graph order to guarantee update installation order. There is an edge from v to w in the write graph if there is an edge from any $\langle P, V\rangle$ in $updates(v)$ to any $\langle Q, W\rangle$ in $updates(w)$ in the installation graph. (Page-oriented operations result in a degenerate write graph, each node of which is associated with the updates of a single variable and with no edges between nodes and hence with no restrictions on installation order of cache pages.)

The write graph is computed from the volatile update buffer by the algorithm WriteGraph(U_v) in Figure 7. In this algorithm, we use the idea of collapsing a graph $\mathcal{A}$ with respect to a partition Π of its nodes. Each set of the partition represents variables that must be written atomically. The result is the graph $\mathcal{B}$ where each node w corresponds to a class π_w in the partition Π and an edge exists between nodes v and w of $\mathcal{B}$ if there is an edge between nodes a and b of $\mathcal{A}$ 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. In the technical report [12], we discuss incremental methods of maintaining the write graph $\mathcal{W}$ so that it evolves as new updates are added to the cache.

procedure PurgeCache
  compute the write graph $\mathcal{W}$
  choose a minimal v node in $\mathcal{W}$
  write operations from the log buffer with updates in $updates(v)$ to the stable log in conflict order
  (this adds these operations to the stable history)
  atomically write values of variables in $variables(v)$ to the stable state
  delete operations with updates in $updates(v)$ from the log buffer
  delete updates in $updates(v)$ from the volatile update buffer
  delete variables in $variables(v)$ from the dirty volatile state
  return

Figure 8: The cache management algorithm PurgeCache.

The cache manager uses the algorithm PurgeCache in Figure 8 to write to the stable state. We can prove that using this algorithm during normal operation and recovery preserves some simple properties of the stable state. For example, if the stable state S is explainable by a prefix σ, then for every update $\langle O, U\rangle$ in the volatile update buffer

1. there are no write-write edges in the volatile history's installation graph from $\langle O, U\rangle$ to updates in σ and
2. every update $\langle P, V\rangle \prec \langle O, U\rangle$ is in σ or in the volatile update buffer.

In the special case that $\langle O, U\rangle$ is a minimal update in the update buffer, these conditions imply that σ can be extended by $\langle O, U\rangle$, so installing $\langle O, U\rangle$ in the stable state will yield an explainable state. These conditions are invariant because operations are added to the update buffer in conflict order during normal operation, and the REDO test skips over updates that violate condition 1 during recovery. We can also prove that these conditions are preserved by the algorithm PurgeCache. Thus, the stable state remains explainable, and the database remains recoverable:

Theorem 3 *PurgeCache preserves the recoverability of the stable database.*

6 Practical Recovery Methods

There are three hard problems that a practical recovery method must solve:

1. **Atomicity**, since multiple pages may have to be installed atomically.
2. **Write Order**, since the cache manager must compute the write-graph dependencies between cached variables in real time.
3. REDO **test**, since this test determines which logged operations are redone during recovery.

Practical methods usually cope with these problems by constraining logged operations to syntactically simple forms. In this section, we discuss common constraints used by existing methods, and propose a less restrictive constraint that preserves most of the simplicity of these methods.

6.1 Existing Methods

Most practical recovery methods permit only *page-oriented operations* that access exactly one page. Such operations yield installation graphs without any write-write edges. This is because if O writes x and P writes x, then P reads either x or nothing at all, so P is in the can redo of O. Consequently, such operations are always installable during recovery, and the REDO test need only test for applicability. Each node of the write graph is associated with a single page, and there are no edges at all between nodes of the graph. This means that the atomicity problem is solved since each page can be installed atomically,[3] and the write order problem is solved since the pages can be installed in any order. Only the REDO test poses a problem.

One example of page-oriented operations is *after-image writes* which have empty read sets and single-page write sets. This single-page write might write to the entire page [3] or to selected records or bytes on the page [2, 4]. These methods differ in the tradeoff they make between log record size and the need to access a page before replaying the operation during recovery. These operations are always applicable, and hence the REDO test can always return true.

Another example is *physiological* operations as described in [6] and used in ARIES [14]. These are state-transition operations that require reading the before-image of a page and then computing its after-image. These operations are *not* always applicable, so the REDO test must test for applicability. This is usually implemented by storing a state identifier in the written page and associating the state identifier in some way with the operation's log record. The address of the log record (called a log sequence number or LSN) is often used as the state identifier [6]. The REDO test simply compares the log record's state identifier with the state identifier stored in the page to determine applicability.

6.2 A New Method: Tree Operations

Understanding installation graphs and their resulting write graphs allows us to generalize page-oriented methods with what we call *tree operations*. A tree operation O reads one page x, and then writes x and possibly other pages in NEW. NEW is the set of pages that have never been written before, and writing a page in NEW removes it from NEW. While O may write multiple pages, the write set of each update U is a single page. We call these tree operations because their conflict graph is a tree.

Like page-oriented operations, tree operations solve the atomicity problem by defining the write set of each update to be a single page. Like page-oriented operations, tree operations yield installation graphs with no write-write edges, so these operations are also always installable and the REDO test need only test for applicability. Testing for applicability is easy to implement using log sequence numbers.

Cache management, however, is more difficult since the write graph is now nontrivial: if O reads x and writes x and y, then there is an edge from y to x in the write graph because there is a read-write edge from $\langle O, y \rangle$ to $\langle O, x \rangle$ in the installation graph since both updates read x. The cache manager needs to do "careful replacement" of these pages; that is, write the pages in a constrained order. These read-write edges in the write-graph are simple to compute, since no cycles ever occur for the second "collapse" to process, so a simple incremental computation of the write graph is possible [12].

Tree operations can improve logging performance for practical problems like splitting the node of a B-tree. Splitting a node requires reading the page containing the old node and splitting its contents by writing back to the old node and a new node, as well as reading and updating the parent node. "B-link-tree" concurrency and recovery techniques [11] link the old and new nodes together. This link preserves the accessibility of the moved data in the new node until the parent is updated. As a consequence, we can log the split of the node and the update of the parent as two separate atomic operations. Let us compare logging the split of the node using page-oriented operations and tree operations:

- Page-oriented operations: Two logged operations are required, one for each node. One is a blind write to the new page requiring that we log the entire contents of this new page. The other is a re-write of the original page, which only requires that we log the split key and instructions to remove all entries greater than the split key from the node.
- Tree operations: One logged operation can deal with the updating of both new and original nodes. This operation reads the old value of the original node and, using the result of this read, writes both original and new nodes. In particular, the new node's contents need not be logged in its entirety because it is derived from the old value in the original node; that is, the entries greater than the split key.

Logging tree operations yields in a smaller log at the cost of requiring "careful replacement:" the value of the new node must be installed in the stable database before the the updated value of the old node can be installed.

[3] Actually, this is only true if we are guaranteed that a disk write does not fail in the middle. When such a failure occurs, media failure recovery must be invoked. For single page writes, a subsequent read can readily detect such a failure.

6.3 Re-cycling Pages

It is important to be able to replenish NEW with pages that have been freed, effectively recycling them and reclaiming space. Recycling an old page, however, can introduce a write-write edge in the installation graph between the last update writing the page prior to its being freed and the first update writing the same page after re-allocation. The operation performing the new update does not read the page, and hence this operation is not guaranteed to be in the *can-redo* of the earlier operation.

Because of these write-write edges, redoing operations in the log during recovery threatens to violate the cache management invariant concerning the absence of write-write edges. This threat can be removed without complicating the REDO test. The trick is simply to take a checkpoint.

We recycle freed pages by *scrubbing* them before reusing them. Scrubbing is done by checkpointing. A checkpoint "cuts" write-write edges from prior incarnations of a freed page by removing from the log all operations that have written to the page x prior to it's being freed. Since x will never be reset to a value prior to the checkpoint during recovery, the page can be added back to NEW. Scrubbing is appropriate for recycling pages in any recovery technique and avoids any change to the technique's REDO test.

6.4 Future Directions

Our framework provides a clean decomposition of the recovery problem that allows us to identify and focus on the three hard problems a practical recovery method must solve: atomicity, the write order, and the REDO test. We hope that the clearer understanding of recovery that we provide here will transform what has been an arcane art into the realm of skilled engineering. We anticipate that this clearer understanding will shortly lead to a number of interesting and practical new recovery.

Acknowledgments

We would like to thank Rakesh Agrawal for urging the first author to de-mystify recovery and Butler Lampson for his enthusiastic interest.

References

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Reading, MA, 1987.

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

[3] K. Elhardt and R. Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9(4):503–525, December 1984.

[4] J. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems—An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978. Also appears as *IBM Research Report RJ 2188*, 1978.

[5] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, and F. Putzolu. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[7] T. Haerder and A. Reuter. Principles of transaction oriented database recovery—a taxonomy. *ACM Computing Surveys*, 15(4):287–318, December 1983.

[8] D. Kuo. Model and verification of a data manager based on ARIES. In *Proceedings of the 4th International Conference on Database Theory*, pages 231–245, October 1992.

[9] D. Lomet. Recovery for shared disk systems using multiple redo logs. Technical Report 90/4, DEC Cambridge Research Lab, October 1990.

[10] D. Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 185–194. ACM, June 1992.

[11] D. Lomet and B. Salzberg. Access method concurrency with recovery. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 351–360. ACM, June 1992.

[12] D. Lomet and M. Tuttle. Redo recovery after system crashes. Technical Report 95/5, DEC Cambridge Research Lab, 1995. To appear.

[13] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufman, 1993.

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

[15] C. Mohan I. Narang and J. Palmer. A case study of problems in migrating to distributed computing: Page recovery using multiple logs in the shared disks environment. Research Report RJ7343, IBM Almaden Research Center, August 1991.

[16] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 337–346, August 1989.

[17] G. Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of the 5th Annual ACM Symposium on Principles of Database Systems*, pages 31–42, March 1986.

[18] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *Proceedings of the 9th Annual ACM Symposium on Principles of Database Systems*, April 1990.