A Theory of Redo Recovery

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

Mark Tuttle
HP Labs Cambridge
Cambridge, MA 02142
mark.tuttle@hp.com

ABSTRACT

Our goal is to understand redo recovery. We define an installation graph of operations in an execution, an ordering significantly weaker than conflict ordering from concurrency control. The installation graph explains recoverable system state in terms of which operations are considered installed. This explanation and the set of operations replayed during recovery form an invariant that is the contract between normal operation and recovery. It prescribes how to coordinate changes to system components such as the state, the log, and the cache. We also describe how widely used recovery techniques are modeled in our theory, and why they succeed in providing redo recovery.

1. INTRODUCTION

After a system crash, redo recovery tries to reconstruct the state of the database at the time of the crash by redoing some subset of the logged operations in some order. There are many techniques for doing this [2, 3, 4, 14], and there has been some work on classifying them [1, 7] and explaining specific techniques [9]. But what are the general principles underlying redo recovery? For redo recovery to work, the state update and recovery processes must cooperate in some fashion. The goal of this paper is to characterize the nature of this cooperation as precisely as possible.

1.1 State Update

Database operations change the state of the database, and state update puts these changes into the stable state. The order in which these updates are made is crucial to the success of redo recovery. Consider the operations

$$A : x \leftarrow y + 1 \qquad \text{and} \qquad B : y \leftarrow 2$$

that read and write variables x and y, both initially 0. Consider Scenario 1 illustrated in Figure 1, in which the operations are invoked in the order A followed by B, and B's changes update the state before the crash, but not A's. Now there is no way for redo recovery reconstruct the value 1 for x simply by redoing one of A or B or both.

The problem is that there is a read-write conflict from A to B in the conflict graph, and updating the state with B's changes before A's violates this ordering. The problem is that B's update of y makes it impossible to redo A to regenerate the value of x. One way to ensure that redo recovery is always possible is to use the conflict graph. If state updates are made in conflict graph order, then redo recovery can replay the remaining operations in conflict graph order and recover the state.

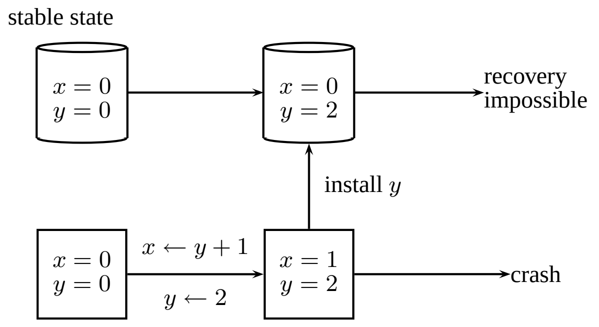

Figure 1: Read-write edges are important.

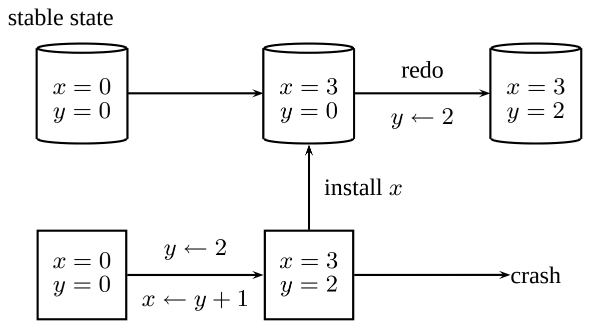

Figure 2: Write-read edges are unimportant.

But state update has more flexibility than this. Consider Scenario 2 illustrated in Figure 2, in which the operations are invoked in the order B followed by A, and A's changes update the state before the crash, but not B's. Now the state can be recovered by redoing B, in spite of the fact that updating the state with A's changes violated the write-read edge from B to A in the conflict graph. A's update to x (which required reading y) does not interfere with B's ability to update y. Hence, the state update order does not need to respect the write-read edges in the conflict graph.

To capture this, we define the *installation graph* to be the conflict graph with the write-read edges removed, and we use this graph to

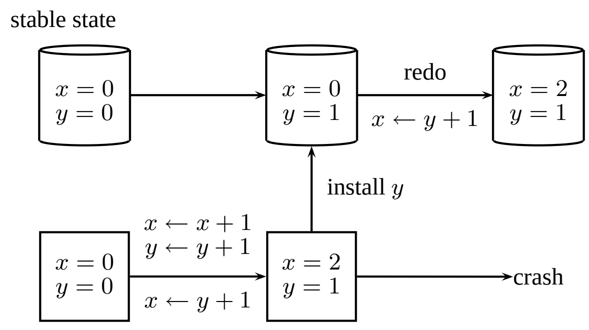

Figure 3: Only exposed variables matter.

guide the state update procedure. We can prove that if state update writes the changes made by operations in an order consistent with installation graph order, then redo recovery can replay the remaining operations in conflict graph order and recover the state. These "installed" operations form a prefix of the installation graph. As long as the stable state can be understood or explained as the result of changes made by the operations in a prefix of the installation graph, redo recovery can recover the state after a crash.

Again, however, state update has more flexibility than this. Consider the operations

$$C : \langle x \leftarrow x + 1; y \leftarrow y + 1\rangle \qquad \text{and} \qquad D : x \leftarrow y + 1$$

that read and write variables x and y, both initially 0. Consider Scenario 3 illustrated in Figure 3, in which the operations are invoked in the order C followed by D, but only C's change to y updates the stable state before the crash, and not x. Notice that redo recovery can recover the state by simply replaying D. There is a read-write edge from C to D due to the variable x, so the installation graph demands that C's changes are made before D's, but there is really only one change made by C that is visible to later operations, and it isn't the value of x. C's change to y is exposed to D because D reads y, but C's change to x is not exposed to any following operation because D overwrites x before any other operation has a chance to read it.

To capture this, we define the notion of an *exposed* variable. Given a set I of operations that have been installed into the stable state and its complement U of uninstalled operations, we say that a variable x is *exposed* if no operation in U reads or writes x (meaning that x already has its final value), or if some operation in U accesses x and the first such operation reads x (meaning x has to have the right value if the system crashes now). We can prove that to install an operation, one need only update its exposed variables. We say that a prefix of the installation graph *explains* the state if each variable x left exposed by the operations in the prefix has the value written by the last operation writing x in the prefix. Our first main result is that if the stable state can be explained by a prefix of the installation graph, then recovery can replay the remaining operations in conflict graph order and recover the state.

1.2 Recovery

Informally, the preceding result says that if a prefix of the installation graph explains the values of the exposed variables, then recovery can recover the state by replaying the remaining operations in conflict graph order. Conversely, if recovery chooses to replay a set of operations, then the remaining operations must form a prefix of the installation graph explaining the values of the exposed variables in order for recovery to succeed. This is our second main result. We capture this requirement with a *recovery invariant*. This invariant says that if, at any time, the recovery procedure would choose to redo some set of operations, then the set of remaining operations form a prefix of the installation graph explaining the state. This is the sense in which state update and recovery must cooperate.

The point of this paper is that the recovery invariant serves as contract between state update and recovery that ensures recoverability. We end this paper with a discussion of how our framework explains logical, physical, and physiological redo recovery techniques, and how these techniques maintain the recovery invariant. We go farther, in fact, and show how focusing on maintaining the recovery invariant allows us to generalize physiological operations to log the split of a B-tree node more efficiently than can be done with conventional physiological operations.

Making the preceding ideas mathematically precise requires some technical machinery. First, theory is traditionally a bit informal about the relationship between operation sequences and conflict graphs, and we find it helpful to nail down this relationship. Second, theory is also somewhat informal about the relationship between between a conflict graph and the changes made to the state by operations in the graph, so we define a state graph to help us talk about state changes. To some extent, this machinery simply lets us prove intuition many already have about recovery. More significant, however, is our formulation of an abstract redo recovery procedure. Recovery actually involves many subtle issues that we have ignored in this introduction, such as interactions among logs, checkpoints, log scans prior to recovery, and the the method of choosing the set of operations to redo during recovery. Finally, we define the notion of a write graph, a kind of state graph derived from an installation graph, and use this write graph to model how real systems accumulate the effects of multiple operations before changing the stable state and installing these operations.

1.3 Prior Work

We have written on recovery theory in the past [12, 13]. Our prior formulations focused primarily on characterizing states from which it is possible to recover. In this work, we focus on the interaction between changing the state and recovery. In particular, our recovery invariant is a precise statement of the contract between the state update process (both during normal operation and during recovery) and the recovery process (and its selection of operations to replay after a crash) in order for these two processes to interact seamlessly. In addition, our work has several other properties:

1. *Simpler*: Our definition of the installation graph is now a simple weakening of the conflict graph. Our prior definition [12] removed write-write edges in addition to write-read edges, but identifying which write-write edges to remove involved an elaborate construction. It turns out that the two definitions are equivalent, in the sense that a state is explainable by a prefix of one iff it is explainable by a prefix of the other.

2. *More precise*: Our definition of the state graph lets us capture precisely our intuitive understanding of the relationships among the state, the cache, and the conflict and installation graphs. State graphs unify how we treat stable state and volatile state, and permit us to consider regimes that maintain multiple versions of variables.

3. *More abstract*: We focus on recovering state without assuming any particular structure for the database implementation, such as how the state is stored on stable or volatile storage,

in contrast to [12]. This abstract characterization can be applied in a number of ways to real systems, as demonstrated in Section 6.

4. *More complete*: We consider general redo tests, instead of restricting attention to redo tests satisfying a particular specification [12]. We also model redo tests and their interaction with the state update process more realistically. Prior formulations [12] modeled the interface between state update and recovery in terms of a particular explanation of the stable state, but we know that real recovery procedures do not explicitly refer to such an explanation at the start of recovery. Real systems fix a redo procedure, and engineer the rest of the system to enforce the recovery invariant with that specific procedure in mind.

The remainder of this paper is organized as follows. In Section 2, we define our model of a database and the notions conflict and state graphs. In Section 3, we define the installation graph and exposed variables, and we prove that an explainable state can be recovered. In Section 4, we define the abstract recovery protocol, describe when it will lead to successful recovery, and state the recovery invariant. In Section 5, we define the write graph and show how the write graph can be used to manage the state so that recovery is possible. Section 6 describes how real systems maintain the recovery invariant, and thus successfully provide recovery. Finally, in Section 7 we discuss some future directions for this work.

2. PRELIMINARIES

We begin with a model of a recoverable system, the definitions of conflict and state graphs, and the notion of an exposed variable.

2.1 System Model

Our system model is quite simple. A full model of a database would require distinguishing between stable state and volatile state, and providing cache managers and log managers to coordinate the movement of information from the volatile state to the stable state. Our notions of a state and changes to this state are indifferent to whether these changes are physically recorded on a disk or in a cache. We study the problem of changing state in a way that allows us to recover the remaining changes if the systems should crash before they are all installed. This model can be extended to a model that more faithfully represents pragmatic implementations of database systems, as we have done [12] and as we intend to do in the future. This model, however, has enough detail to allow us to explain how state changes and recovery are coordinated.

A recoverable system has a set of variables and a set of values they can assume. A system state describes the values of the variables at a given point in time. A program accesses or changes the state by invoking system operations, and these changes are recorded in the log with logged operations. These two sets of operations can be quite different, and the only operations of interest to us are the logged operations. Each operation atomically reads a set of variables and then writes a set of variables. These ideas can be modeled mathematically as follows.

Fix a set of *variables* and a set of *values*. A *state* is a function that maps each variable to a value. An *operation* is a function with a fixed set of input variables and a fixed set of output variables. We call these sets of input and output variables the *read set* and *write set*, respectively. An *operation sequence* is a sequence $O_1O_2\dots O_k$ of operations. A *state sequence* is a sequence $S_0S_1S_2\dots S_k$ of states generated by an operation sequence $O_1O_2\dots O_k$, where S_0 is an initial state and each S_i is the result of applying O_i to S_{i-1}. The state sequence generated by an operation sequence obviously depends on the initial state S_0, but the value of S_0 is usually clear from context or unimportant, and we usually omit reference to it.

In the next two sections, we define directed graphs to model operation sequences and state sequences. Given a directed graph, the *predecessors* of a node n is the set of all nodes m such that there is a path from m to n in the graph. A *prefix* of a directed graph is a subgraph induced by a set of nodes having the following property: If a node is in the prefix, then all of its predecessors are in the prefix.

2.2 Conflict Graph

An operation sequence generates a conflict graph defined as follows. For an operation O, the *preceding write* to x is the preceding operation W that writes x such that there is no operation writing x between W and O. Similarly, the *following write* to x is the following operation W that writes x such that there is no operation writing x between O and W. The *conflict graph* generated by an operation sequence is a directed, acyclic graph with each node labeled with an operation. There is an edge from O to P if they conflict in one of the following ways:

- *write-write conflict*: O writes x and P writes x and O is P's preceding write.
- *write-read conflict*: O writes x and P reads x and O is P's preceding write.
- *read-write conflict*: O reads x and P writes x and P is O's following write.

We assume that operations labeling nodes are distinct, so we can refer to a node by the operation labeling it.

Conversely, a conflict graph generates a set of operation sequences obtained by totally ordering the operations labeling the graph. The following result states that each of these operation sequences can be used to generate the conflict graph.

Lemma 1: If σ is any total ordering of the operations labeling a conflict graph $\mathcal{C}$, then $\mathcal{C}$ is the conflict graph generated by σ.

One consequence of this lemma is that we can model a log as a set of operations ordered only by the conflict graph. It is not necessary to have a totally ordered log reflecting the exact execution order of the operations. Only conflicting logged operations need to be ordered.

2.3 Exposed Variables

Conflict graphs help specify what values variables should have during recovery. The recovery process *replays* a subset of the operations in conflict graph order, bypassing others which are considered installed. If x contains the wrong value at the start of recovery, then an operation replayed during recovery must write to x to give it an appropriate value before any other operation replayed during recovery reads x. Then the current value of x will never be observed and can be considered *unexposed*. Otherwise the current value of x is *exposed* because an operation reads it during recovery or it is needed following recovery. Notice that, given a subset of the operations in a conflict graph that read or write x, the conflict graph induces a partial order on these operations, so the notion of a minimal such operation is well-defined.

Let $\mathcal{C}$ be a conflict graph and let I be a subset of the operations labeling $\mathcal{C}$. For example, I might be the operations labeling a prefix of $\mathcal{C}$, representing the set of installed operations. A variable x is *exposed* by I if

- no operation outside of I accesses x, or

- some operation outside of I accesses x, and a minimal such operation reads x,

and x is *unexposed* otherwise. In particular, a variable x is unexposed by I if some operation outside of I accesses x, and a minimal such operation writes x without reading x.

The conflict graph $\mathcal{C}$ grows as operations are performed during normal execution. The set I of installed operation grows as updates made by these operations are written into the state. If the conflict graph $\mathcal{C}$ grows and the installed set I does not, then the status of a variable x may flip from exposed to unexposed, but once it becomes unexposed by I, it remains unexposed. On the other hand, if the installed set I grows and the conflict graph $\mathcal{C}$ does not, then the status of a variable x can flip back and forth between exposed and unexposed.

2.4 State Graphs

A state graph is an abstract representation of a state sequence, just as a conflict graph is an abstraction of an operation sequence. The conflict graph tells us how the changes made by operations in the graph interact, but it does not tell us what these changes are. A state graph models the evolution of the state over time. We show that two operation sequences that have the same conflict graph also have the same conflict state graph.

A *state graph* is a directed, acyclic graph satisfying the following properties:

- Each node n is labeled with
  - a set *ops*(n) of operations, and
  - a set *writes*(n) of variable-value pairs $\langle x, v\rangle$, at most one pair $\langle x, v\rangle$ for each variable x, and only pairs $\langle x, v\rangle$ for variables x written by operations in *ops*(n).
- For each pair of nodes m and n, if *writes*(m) and *writes*(n) both contain a variable-value pair for variable x, then there is a path from m to n or from n to m.

We assume that the sets of operations labeling the nodes are disjoint. We say that node n *writes* v *to* x if the set *writes*(n) labeling n contains the pair $\langle x, v\rangle$. We define *vars*(n) to be the set of variables x such that $\langle x, v\rangle \in$ *writes*(n) for some value v.

An operation sequence generates a state graph as follows. Given an initial state S_0, the *state graph* $\mathcal{S}$ *generated by* $O_1 \cdots O_n$ is obtained by relabeling each node n of the conflict graph generated by $O_1 \cdots O_n$ as follows:

- the set *ops*(n) is the singleton set $\{O_i\}$ where O_i is the operation labeling n in the conflict graph, and
- the set *writes*(n) is the set of all variable-value pairs $\langle x, v\rangle$ such that x is in O_i's write set and v is the value of x in S_i, where $S_0 S_1 \cdots S_n$ is the state sequence generated by $O_1 \cdots O_n$.

Notice that the value of x in S_i is the value that O_i writes when it is performed in state S_{i-1}. Since the operations labeling the conflict graph are distinct, the same is true of the state graph, so we can again refer to nodes by the operations that label them.

In the preceding definition, an operation sequence $O_1 \cdots O_n$ was used to generate a state sequence $S_0 S_1 \cdots S_n$, and the state sequence was used to generate a state graph $\mathcal{S}$. We can also use the state graph $\mathcal{S}$ to recover the states in the sequence $S_0 S_1 \cdots S_n$. The natural state defined by a state graph $\mathcal{S}$ maps each variable x to the last value written to x by any node in the graph. Since the definition of a state graph requires that the nodes writing x are totally ordered, the last value written to x is well-defined. Of course, there may be variables that are never written by any node in $\mathcal{S}$ or by any operation in the sequence $O_1 \cdots O_n$ used to generate $\mathcal{S}$. Values for these variables must come from the initial state S_0 in the sequence $S_0 S_1 \cdots S_n$ used to generate $\mathcal{S}$. Formally, given an initial state S_0, the *state determined by a state graph* $\mathcal{S}$ maps a variable x to

- a value v, where $\langle x, v\rangle \in$ *writes*(n) and n is the last node in $\mathcal{S}$ with $x \in$ *vars*(n), or
- to the value of x in S_0 if there is no node n with $x \in$ *vars*(n).

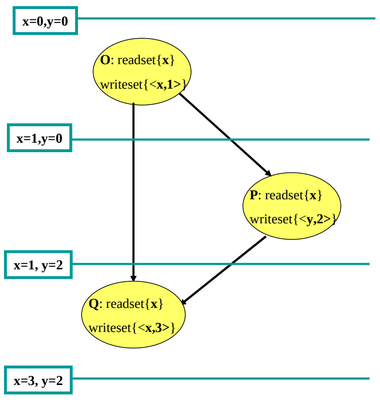

Figure 4: Conflict state graph

Since a prefix of a state graph is a state graph, every prefix of the state graph $\mathcal{S}$ determines a state, and the next lemma shows that we can reconstruct the state S_i in the sequence $S_0 S_1 \cdots S_n$ used to generate $\mathcal{S}$ by considering the subgraph of $\mathcal{S}$ induced by the operations $O_1, \ldots, O_i$.

Lemma 2: If the operation sequence $O_1 \cdots O_n$ generates a state sequence $S_0 S_1 \cdots S_n$ and a state graph $\mathcal{S}$, then S_i is the state determined by the prefix of $\mathcal{S}$ induced by the operations $O_1, \ldots, O_i$.

In fact, we can prove that any state determined by any prefix of this state graph is reachable by any total ordering of the operations labeling that prefix.

Finally, the state graph generated by an operation sequence depends only on the conflict graph of the operation sequence. The crucial observation is that the state graph depends only on the order of conflicting operations. Since the conflict graph characterizes the order of conflicting operations, the conflict graph uniquely determines a state graph. In light of this, we say that the *conflict state graph* for a conflict graph $\mathcal{C}$ is the state graph generated by any operation sequence generating $\mathcal{C}$, since they are all the same. We call the state determined by the entire conflict graph the *final state*. Our goal will be to recover the final state.

Figure 4 illustrates the conflict graph and its state graph produced by the sequence of operations O (reading x and writing x), P (reading x and writing y) and Q (also reading x and writing x). The solid lines separating the operations indicate the system states determined by the prefixes that the lines identify. The values of the variables of these states are given in the rectangles associated with the lines.

Most of the results in this section are intuitively clear or are things that one might expect to be true, but we have established them from first principles. Recovery is essentially the problem of reconstructing state without knowing much about the way the state was originally constructed, and state graphs tell us how much flexibility we have when reconstructing the state.

3. POTENTIAL RECOVERABILITY

In this section, we identify the states that can be recovered by replaying operations. Given a conflict graph, we say that a state is *potentially recoverable* if we can replay some subset of the operations in the conflict graph from this state in conflict graph order and end up with the final state determined by the conflict graph.

3.1 Installation Graph

What sets of operations can appear as installed in a potentially recoverable state? One trivial answer: if a state contains the operations labeling a prefix of the conflict graph, then we can replay the remaining operations in conflict graph order and recover the state. However, this is not the only answer. In Scenario 2 in the introduction, we saw a conflict graph with a write-read edge from B to A, yet we can take a state with A installed and recover the final state by replaying B. The state "containing" A is potentially recoverable, yet the set $\{A\}$ does not determine a prefix of the conflict graph. However, $\{A\}$ does determine a prefix of a graph we call the installation graph. Prefixes of the installation graph include the prefixes of the conflict graph. Their sets are the sets of operations that can appear in potentially recoverable states.

The *installation graph* is the subgraph of the conflict graph obtained by removing the edges resulting solely from write-read conflicts. In the example of Figure 4, the conflict graph contains an edge from O to P that denotes a write-read conflict. Hence, the installation graph for the example in Figure 4, which is shown in Figure 5, does not have an edge from O to P. Only the edges from O to Q (a write-write and read-write conflict) and from P to Q (a read-write conflict) remain. The removed conflict graph edge is shown by a dotted arrow in Figure 5. It is easy to see that the operations labeling a prefix of the conflict graph induce a prefix of the installation graph, but not vice versa.

Every prefix of the installation graph induces a prefix of the installation state graph labeled with the same operations. The *state determined by a prefix of the installation graph* is the state determined by the corresponding installation state graph prefix. This state contains the final values for all variables written by the operations in that prefix (when the operations are executed in conflict graph order). In Figure 5, the conflict graph edge from O to P does not appear in the installation graph and there is an additional recoverable state that can be identified, denoted by the dashed line separating operation nodes P from nodes O from Q.

3.2 Explainable States

Does a potentially recoverable state need to contain values for all of the variables written by the operations in a prefix? Not necessarily. In particular, values of unexposed variables are irrelevant. We say that a variable x is *exposed* by a prefix of the installation graph if x is exposed by the set of operations labeling the prefix, and x is *unexposed* otherwise. We say that a prefix σ of the installation graph *explains* a state S if the exposed variables have the same value in S and the state determined by σ. We call states that are explained by a prefix of the installation graph *explainable*. An unexposed variable need not contain any specific value. We will show that explainable states are potentially recoverable.

3.3 Replaying Operations

Consider just the first step of recovering a state, namely replaying the first operation whose effects do not appear in the state. We say that an operation O is *applicable* to a state S if the values of variables in O's read set are the same in S and the state determined by O's predecessors in the conflict graph. This means that O will read the same values in S as it did in any execution represented by

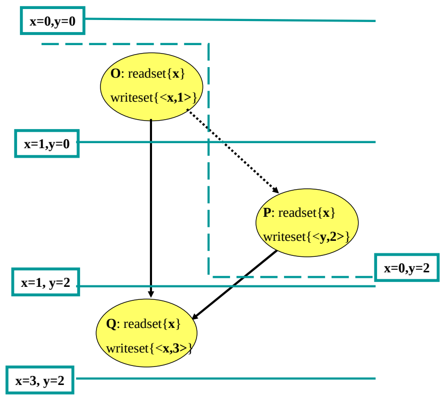

Figure 5: Installation state graph illustrating the removal of write-read edges.

the conflict graph, so it will write the same values. An operation O is said to be *installed* in a state S relative to a particular prefix σ that explains S if O is in σ, and is *uninstalled* otherwise. A *minimal uninstalled operation* after σ is a minimal operation in the conflict graph that is not in σ. For example, in Figure 5, the minimal uninstalled operation after O is P, while the minimal uninstalled operation after P, a prefix permitted by the installation graph, is O. Note that in all cases, the minimal uninstalled operation sees exactly the same values for its read set, e.g. operation O sees $x = 0$ even when operation P is installed before it, as it does when operations are executed in conflict graph order.

We denote by $\sigma; O$ the extension of a prefix σ by a minimal uninstalled operation O, and we denote by $S; O$ the state obtained by applying O to S. We can show that if S is a state explained by σ and if O is a minimal uninstalled operation after σ, then O is applicable to S and $S; O$ is a state explained by $\sigma; O$.

3.4 Potentially Recoverable States

It should be clear that an explainable state is potentially recoverable. If we can identify an installation graph prefix explaining the state, then we can recover the final state by replaying uninstalled operations in conflict graph order.

Theorem 3 (Potential Recoverability Theorem): If S is a state explained by a prefix σ of the installation graph, then S is potentially recoverable.

PROOF. We proceed by induction on $m \geq 0$, the number of uninstalled operations.

Suppose $m = 0$. Since there are no uninstalled operations, σ contains every operation in the conflict graph. Since σ explains S and since every variable is exposed, each variable x in S has the value written by the last operation writing x in σ and hence in the conflict graph, so each variable x has the same value in S and the state determined by the conflict graph, so S is the state determined by the conflict graph.

Suppose $m \geq 0$ and the induction hypothesis holds for $m - 1$. Since O is a minimal uninstalled operation, O is applicable to S. Further, $\sigma; O$ is a prefix of the installation graph that explains $S; O$. Since the number of uninstalled operations after $\sigma; O$ is $m - 1$, the induction hypothesis implies that replaying these uninstalled operations against $S; O$ in any order consistent with the conflict graph yields the state determined by the conflict graph. Since replaying O and then replaying these operations amounts to replaying the operations uninstalled after σ in an order consistent with the conflict graph, we are done. $\square$

4. RECOVERY

So far we have characterized the states from which we can recover. Now we focus on the recovery process itself. We describe an abstract recovery procedure and what is required for this procedure to work. A recovery procedure begins with the state and the log as of the time of the crash. The recovery procedure scans the log and examines each log record in turn. It determines whether the operation mentioned in this record should be replayed, and replays the operation if the answer is yes. When the recovery procedure reaches the end of the log, it has rebuilt the system state, and it terminates. The heart of this procedure is the *redo test* that recovery uses to determine whether a logged operation should be replayed.

During normal operation, the system may take a *checkpoint*. A checkpoint procedure updates the state and the log so as to identify a point in the log prior to which the effects of logged operations are reflected in the state. If the system crashes at a later point, the recovery procedure need only examine the part of the log following this checkpointed log prefix instead of starting at the beginning of the log.

Some recovery procedures have an *analysis phase* in which they scan the log to find information needed by the redo test to determine whether an operation should be replayed. An analysis phase usually happens at most once, at the start of recovery.

We model each of these aspects of recovery as follows.

4.1 The Log

In practice, a log is a linear sequence of records, each record containing information about the invocation of an operation, together with additional information about this operation and its invocation, with log records maintained in invocation order. Given a conflict graph $\mathcal{C}$, we define a *log* for $\mathcal{C}$ to be any directed, acyclic graph with the following properties:

- Logged operations are from the conflict graph: Each node (record) is labeled with an operation and possibly some other labels, and the set of operations labeling the conflict graph and the log are the same.
- Log order is consistent with the conflict order: If there is a path from O to P in the conflict graph, then there is a path from O to P in the log.

4.2 The Checkpoint

There are many ways that systems implement a checkpoint. It might be as simple as flushing the log to disk and writing a special checkpoint record to the end of the log, indicating where the recovery procedure should begin, or it could be much more complicated. No matter how the checkpoint is implemented, its function is to identify a set of operations (or log records) that the recovery procedure can ignore. We assume that the recovery procedure is called with a set of operations that have been checkpointed. The checkpointed log records usually constitute a prefix of the log, but that is not required. What is required is that the “checkpointed” log records denote operations that do not need to be replayed during recovery (because these operations are installed).

4.3 The Analysis

We permit the analysis phase of recovery to be arbitrary. We assume the recovery procedure has a function *analyze* that performs the analysis phase and returns some value called the *analysis*. This value might be a simple log position or a complicated data structure. In the simple case, the analysis function might map the state and the log at the start of recovery to a position in the log for the start of recovery (the position of the last checkpoint record, for example). In a more complex case, an analysis phase might be performed once for each operation before invoking the redo test on this operation, e.g. to determine the set of currently unrecovered operations and even the result of the previous analysis phase. To cover this range of options, our model permits the *analyze* function to map a state, a log, a set of operations (the unrecovered operations), and an earlier analysis to another analysis.

```
procedure recover(state, log, checkpoint)
  unrecovered = operations(log) - checkpoint
  analysis = null
  while unrecovered is not empty
    O = minimal operation in unrecovered
    analysis = analyze(state,log,unrecovered,analysis)
    state = if redo(O,state,log,analysis)
      then O(state)
      else state
    unrecovered = unrecovered - {O}
  end while
```

Figure 6: A redo recovery procedure.

4.4 The Recovery Procedure

The heart of the recovery procedure is a *redo test* that returns true or false, given an operation, a state, a log, and an analysis. For a redo test *redo* and an analysis function *analyze*, we define the redo recovery procedure to be the algorithm *recover* given in Figure 6. The recovery procedure is called with a state, a log, and a checkpoint at the time of the crash. It begins by setting the unrecovered operations to the logged operations not appearing in the checkpoint and then considers the unrecovered operations in log order. For each operation O, the procedure goes through an analysis phase, then it invokes the redo test to determine whether the state should be updated by replaying O. This procedure has an analysis phase at the start of every iteration of the loop. The typical case where there is a single analysis phase at the start of recovery is captured when the analysis phase is invoked with the initial null value and is the identity function otherwise.

Given a state, a log, and a checkpoint, the recovery procedure replays some subset of the operations

$$operations(log)$$

in the log. Let U be the set of (uninstalled) operations for which the redo test returned true. Then the set I of remaining operations are the *installed* operations. For every variable v appearing in the recovery procedure, let v_i be the value of v at the end of the ith iteration of the main loop, and let v_0 be the initial value of v at the start of the first iteration. Let

$$\textit{redo\_set} = \{O_i : redo(O_i, state_{i-1}, log_{i-1}, analysis_i)\}$$

be the set of operations replayed during the execution; that is, the set of operations O_i such that on the ith iteration of the main loop the redo test returned true when applied to O_i. Let

$$installed_i = operations(log) - (\textit{redo\_set} \cap unrecovered_i).$$

be the set of all logged operations that will not be redone after the ith iteration of the loop, and hence can be considered installed.

We can now prove the procedure *recover* is correct:

Corollary 4 (Recovery Corollary): Given a state, a log, a checkpoint, and an execution of the procedure *recover*, if the set of *installed* operations $operations(log) - \textit{redo\_set}$ induces a prefix of the installation graph that explains $state$, then *recover* terminates with the state determined by the conflict graph.

PROOF SKETCH. It follows by a simple inductive argument that $installed_\ell$ is a prefix that explains $state_\ell$ at the end of each iteration ℓ. In particular, the recovery procedure terminates with a state explained by the entire installation graph. This means that the exposed variables in this final state have the same value in this state and the state determined by the entire installation graph. Since all variables are left exposed by the entire installation graph, and since the state determined by the installation graph and the conflict graph are equal, the recovery procedure terminates in the state determined by the conflict graph. □

4.5 Keeping Systems Recoverable

We have just shown that the recovery procedure will recover the system state as long as the following is true:

Recovery Invariant

The set $operations(log) - \textit{redo\_set}$ induces a prefix of the installation graph that explains the state.

Supporting recovery is difficult because the truth of this invariant depends on every system component. The state is what needs to be explained. The log is where the operations are recorded. The checkpoint and the log initialize the set of unrecovered operations in the recovery procedure. The redo test determines which of these unrecovered operations are replayed, and hence what prefix of the installation graph must explain the state. Maintaining this invariant requires that every change to the state be accompanied by a corresponding change to the set of operations that the redo test choses to redo.

5. MANAGING SYSTEM STATE

If a system updates the stable state in installation graph order, then the stable state can always be recovered. However, real systems don't update the state with changes made by operations just one operation at a time. When they move pages from the cache to the disk, a page frequently contains changes made by a number of operations. This can make installing operations a bit tricky.

Consider the operations

$$E : x \leftarrow y + 1 \quad \text{then} \quad F : y \leftarrow x + 1 \quad \text{then} \quad G : x \leftarrow x + 1$$

After executing these operations, suppose the system updates the stable state with the value of x in an attempt to install E and G. Such an update violates a read-write installation graph edge between F and G. Similarly, we cannot update the stable state with the value of y, as that update would violate a read-write installation graph edge from E to F. Stable state is unrecoverable if either x or y is updated singly, since we can't recover the other value by replaying any combination of the operations. The system has to update the stable state with the values of both x and y atomically, hence installing the operations E, F, and G atomically.

Accumulating the changes from multiple operations can require the system to update sets of variables atomically. Multi-variable write sets can also require this. However, we can take advantage of unexposed variables to keep these sets of variables from getting too large.

Consider the operations

$$H : \langle x \leftarrow x + 1; y \leftarrow y + 1 \rangle \qquad \text{then} \qquad J : y \leftarrow 0$$

The fact that H writes both x and y appears to force us to update x and y atomically to install H. Further, if we accumulate the changes in y, it would appear that we need to install H and J atomically. However, we notice that J's blind write to y makes y unexposed after H. It follows that we need only update the stable state with the value of x, for us to consider H installed, since y's value is unexposed and does not need to be explained.

As these examples illustrate, a system has a great deal of flexibility in how to update the stable state. We now define a *write graph* and operations on this graph that capture this flexibility. The write graph tells the system what sets of operations must be installed atomically, and what updates to the stable state are required in order to install them. We prove that if the system respects the write graph, then the stable state can always be explained by a prefix of the installation graph, and hence that the stable state can always be recovered.

5.1 Write Graph

A *write graph* is a state graph where each node is labeled with a boolean variable *installed* such that the nodes with *installed* set to true form a prefix of the write graph, and where the underlying state graph is obtained from an installation state graph by applying the following operations:

- *Install a node*: Set the boolean variable *installed* labeling a node n to true. Every predecessor of n must have *installed* set to true.

- *Add an edge*: Add a directed edge from a node n to a node m. The node m must have *installed* set to false, and the resulting graph must be acyclic.

- *Collapse nodes*: Replace a set S of nodes with a single node n. The resulting graph must be acyclic. For node n, *ops*(n) is the union of *ops*(s) for all $s \in S$ and *writes*(n) is the set $\langle x, v \rangle$ for which there is an $s \in S$ satisfying: (i) $\langle x, v \rangle \in$ *writes*(s) and (ii) if $\langle x, v' \rangle \in$ *writes*(t) for some $t \in S$ then t is ordered before s in the old graph. There is an edge from m to n in the new graph if there is an edge from m to some $s \in S$ in the old graph, and an edge from n to m in the new graph if there is an edge from some $s \in S$ to m in the old graph. The value of *installed* at n is true iff any node in S has *installed* set to true.

- *Remove a write*: Remove a pair $\langle x, v \rangle$ from the set *writes*(n) labeling a node n. For every node m reading x, either m has *installed* set to true, or m is ordered before n and a node following n writes x without reading it. (For example, m could be ordered before n due to an edge resulting from the *Add an edge* operation).

The simplest write graph is the installation state graph where each node corresponds to an installation graph node. If the installation graph node is labeled with operation O, then the write graph node is labeled with the variable-value pairs that O writes. A system can keep the state potentially recoverable by installing operations in installation graph order. This corresponds to updating the state by choosing nodes of the write graph in write graph order, and atomically updating the state with the values labeling the node.

A system might choose to constrain the order of updates more than they are constrained by an installation graph state graph by adding an edge to the state graph with the *Add an edge* operation. Further, most systems do not maintain multiple versions of a page in cache, but keep a single copy that reflects the effects of all recent operations on the page. The system can enforce a single copy of a

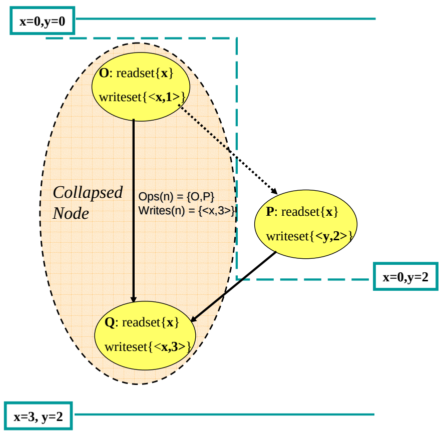

Figure 7: Write graph showing the collapse of nodes writing to x.

page by merging the cache nodes of the write graph that write to the page with the *Collapse nodes* operation. Collapsing an uninstalled node with an installed node is frequently the way in which systems update the stable system state. If operations can write to multiple variables (pages), then merging write graph nodes that write a common variable can lead to a single write graph node writing a larger number of variables than any operation does on its own [12], requiring atomic writes to large portions of the state. We can sometimes avoid writing a variable (page) x updated by an operation of the write graph by removing the update to x from a node with the *Remove a write* operation. This is possible exactly when there is no uninstalled node that reads the value of x being removed.

The operations on write graphs guarantee that a prefix of the write graph corresponds to a prefix of the installation graph from which it is derived, and the state determined by a prefix of a write graph is explained by the corresponding prefix of the installation graph. Thus, if we update the state in write graph order, then the state remains potentially recoverable:

Corollary 5: The state determined by a prefix of a write graph is potentially recoverable.

The identification of the states of write graph prefixes as states of installation state graph prefixes makes it obviously true that cache managers exploiting write graphs keep the system state explainable, and hence potentially recoverable.

Figure 7 illustrates a write graph derived from our running example of Figure 4 in which all operations writing to a common set of variables are collapsed. That results in operations O and Q being collapsed, making some recoverable states "inaccessible". Note in particular, that the edges of this write graph make it clear that the cache manager needs to write y into the state prior to writing x, since operation P needs to be installed prior to the collapsed operations O and Q.

6. REAL RECOVERY METHODS

We now turn our attention to how actual recovery methods maintain the recovery invariant. To demonstrate the generality of our work, we consider three generic log-based recovery technologies — logical, physical, and physiological recovery [6] — and note that every log-based redo recovery method used today that we are aware of fits into one of these categories. These methods all update the state and make other changes that have the effect of updating *redo_set*, and they make these changes in a way that updates the state and *redo_set* atomically. Since these changes are made atomically, the recovery invariant is maintained, and the *recover* procedure can always recover the state after a crash.

In all recovery methods described below, stable state is represented by a single write graph node, the initial or minimum node of the write graph. This node is the result of collapsing into a single node all of nodes of the state graph labeled with operations that are installed in the stable state.

6.1 Logical Recovery

A logical operation may be thought of as a mapping from one database state to the next. At least in concept, the entire database (all its pages) may be read, and the entire database may be written. Hence, we need to atomically transform one database state, in its entirety, to a successor database state. This appears difficult, but in fact this kind of transformation was done previously by System R [5], though its operations were not actually "logical" operations.

In System R, system stable state on disk is unchanged between checkpoints. Pages updated since the last checkpoint are maintained partially in a main memory cache and partially in a disk staging area. Pages in the cache contain the most recent updates to the pages, more recent than the updates in the staging area. The system periodically stops normal execution (*quiesces*). Prior to quiescing, the staging area pages do not constitute a write graph node. During the quiesce, the more recently updated cached pages are written to the staging area, either as new pages or as updates to existing pages. At this point, the staging area becomes the second node of a two node write graph, the other node being the stable state. (Alternatively, during normal system operation, the updates labeling this second write graph node are split between the cache and the staging area, and it is only after quiescence, after the cached updates have been flushed to the staging area, that the staging area itself contains all of the updates labeling this second node.)

A checkpoint record is then appended to the log, indicating that the staging area pages now replace prior versions of these pages in the installed system state. Writing this checkpoint record "swings a pointer" that atomically installs into stable state all operations logged since the previous checkpoint. This collapses the two write graph nodes into a single node.

After a crash, the recovery procedure starts with the stable system state identified in the last checkpoint record. It replays all subsequent log operations against that version. Writing the checkpoint record, therefore, simultaneously installs operations (collapses write graph nodes) and removes these operations from *redo_set* by moving them to the checkpoint set. Since the change to state and the change to *redo_set* are done atomically, the recovery invariant is preserved.

6.2 Physical Recovery

Early recovery techniques frequently exploited physical recovery, logging the exact bytes of data and the exact locations written by the logged operations. Physical operations do not read data, they only write. So there are no write-read or read-write conflicts in the installation graph, only write-write conflicts. Both whole and partial page logging have been used [1]. Like logical recovery, all operations logged since a last checkpoint record on the log are replayed during recovery

Stable system state is represented by the minimum write graph node. The write graph suffix represents the main memory cache holding the accumulated changes not yet reflected in the stable state. In this write graph suffix, there is at most one node writing to any variable, since updates to a page are accumulated in the cache on a page-by-page basis. The installation graph and corresponding state graph consist of chains of nodes, one chain for each page consisting of all operations that access that page. The write graph, which is formed by collapsing all nodes with operations writing the same page, is an initial node followed by a single write graph node for each page.

As with logical logging, checkpointing has the effect of installing operations. But now this is much simpler. The checkpoint process shifts log operations from *redo_set* to the checkpoint set. While these operations are in *redo_set*, the variables they update are unexposed, since physical operations do not read variables. Hence, values of these variables are irrelevant to recovery, so we can set them to whatever values are convenient. Prior to writing the checkpoint record, therefore, we set them to the values they possess in the cache (which includes the effects of the moved operations). Writing the checkpoint record therefore has the effect of atomically removing these operations from *redo_set* and, since their effects are already present in the stable state, installing the operations. This atomic installation preserves the recovery invariant.

6.3 Physiological Recovery

A physiological operation reads and writes exactly one page. It identifies the page by a "physical" page identifier, but performs a "logical" operation on that page. The operation's log record position is called its log sequence number (*LSN*). Each page of the system state is tagged with the LSN of the last operation that updated it. The LSN is usually on the page, but there are other ways of tagging pages [13]. LSNs increase monotonically with each new operation. Each update operation on the page sets the page LSN to its LSN.

Stable state is again represented by the minimum write graph node. The write graph suffix is the same as for physical recovery. Since operations read and write at most one page, there are edges from the initial node to the subsequent write graph nodes, but there are no edges among these subsequent nodes. Consequently, all of these subsequent nodes are uninstalled minimal nodes, and the system is free to install their operation sets in any order.

All the operations of the write graph node are installed into the state atomically when its page is written to disk. This atomic installation is modeled by collapsing a minimal node of the write graph into the initial node. This has the effect of updating the LSN of the page in the initial node with the LSN of the last operation writing to the page.

After a crash, the recovery procedure scans the log. For each operation, it compares the LSN tagging the page with LSN of the operation. If the page LSN is at least as high the operation's LSN, then the operation is already installed and is bypassed during recovery. It follows that the LSN updating that occurs when write graph nodes are collapsed also has the effect of updating the set *redo_set* of operations replayed during recovery. Since the change to the state and *redo_set* is atomic, the recovery invariant is maintained.

6.4 Generalized LSN Based Recovery

While the restriction of physiological operations to read and write exactly one page simplifies cache management, it is possible to exploit LSNs to deal with a broader class of operations. As with physiological operations, we associate the LSN of an updating operation with each variable (page) in its write set when the operation is executed. Thus, as before, the page LSN denotes the last operation that updated it. Since we are writing the updates of a write

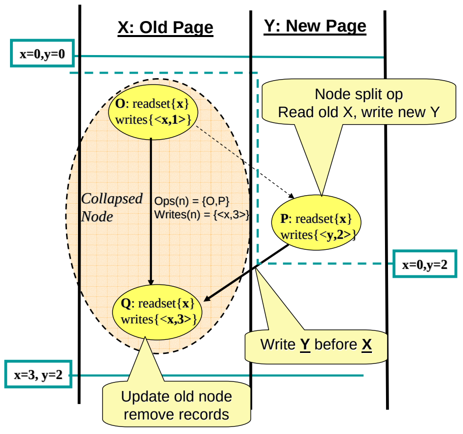
X: Old Page
Y: New Page
x=0,y=0
O: readset{x} writes{<x,1>}
Node split op Read old X, write new Y
Collapsed Node
Ops(n) = {O,P} Writes(n) = {<x,3>}
P: readset{x} writes{<y,2>}
x=0,y=2
Q: readset{x} writes{<x,3>}
Write Y before X
x=3, y=2
Update old node remove records

Figure 8: Write graph showing the effect of using log operations that read and write different variables to produce a B-tree split.

graph node atomically, all variables in an installed operation's write set must be tagged with an LSN that is at least as large as the LSN of the operation. If an operation is uninstalled, all variables of its write set will have LSNs that are less than the operation's LSN.

Exploiting log operations that can read and write different variables is potentially very useful for database recovery [11]. A log operation might deal with node splitting in a B-tree by reading the old full B-tree page and writing a new page with half the contents of the old node. Such an operation avoids "physically" logging the half of a splitting B-tree node that is used to initialize the value of a new node, which is required when physiological operations are used. A subsequent operation then removes the "moved" half of the contents from the old node. The cache manager must be sure to enforce that the new B-tree node is written before the old node is over-written by the operation that removes the moved contents and completes the split. The write graph of Figure 8 illustrates exactly this. Operation P has the form of the operation that reads the old page x and writes the new page y, while operation Q has the form of the operation writing the old page x to remove the half of its contents that were moved to new page y. This results in the write graph edge from the node for P to the collapsed node for O and Q. This edge provides for exactly the enforcement of installation graph order of a non-trivial sort by the cache manager. This requirement for "careful" write order flows directly from our recovery theory.

7. DISCUSSION AND DIRECTIONS

Our theory allows us to talk about the system state and accumulated changes to the state, and to talk about installing some of these changes and recovering the remaining changes after a crash. With this, we can precisely state and prove the contract that must exist between normal execution when state is updated and recovery when operations are replayed. This contract is our recovery invariant. We have shown how most common recoverable systems are describable in our model and how they enforce this recovery invariant. Using this understanding, one future direction is to define new classes of logged operations having recovery methods with potential advantages over current methods, especially when extending recovery to new areas [10, 13].

Another direction is to make explicit more of the detail and structure present in real database systems. One example is the existence of the volatile and stable versions of the state and log, and how the cache manager and log manager coordinate advancing stable database state with the movement of volatile log records to the stable log:

- A problem arises when operations write multiple variables, requiring the system to make atomic changes to multiple variables in the state. How to manage or avoid large atomic transitions is challenging.
- The write-ahead log protocol requires an operation's log record be forced to disk before the operation's effects are written to disk. Exploiting unexposed variables to reduce writes to system state can require the log manager to be more conservative and flush log records even earlier [12].

There are recovery possibilities not covered by our theory. Our theory says that if we begin with a state explained by a prefix of the installation graph and replay the uninstalled operations in conflict graph order, then each operation is applicable when replayed, i.e., recovery reads the same values as read during normal operation, so the values written are also the same. There have been interesting examples in which operations can be replayed even then they are not applicable and write different values during recovery. The key is that these writes are to the unexposed portion of the state, and hence the values written are irrelevant [13].

The current theory, even without these elaborations, demonstrates fundamental principles of recovery. Most important, it captures the need to make sense of the values in the stable state in terms of what operations are considered installed, and the need to keep this explanation consistent with the set of operations the recovery procedure will consider uninstalled during recovery.

8. 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 TODS*, 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] V. Kumar, M. Hsu (eds). *Recovery Mechanisms in Database Systems*. Prentice Hall, 1998.

[9] D. Kuo. Model and verification of a data manager based on ARIES. In *ICDT 1992* 231–245.

[10] D. Lomet. Persistent Applications Using Generalized Redo Recovery. *ICDE 1998* 154–163.

[11] D. Lomet. Advanced Recovery Techniques in Practice. in *Recovery Mechanisms in Database Systems* Prentice Hall, 1998

[12] D. Lomet and M. Tuttle. Redo recovery after system crashes. *VLDB 1995* 457–468

[13] D. Lomet and M. Tuttle. Logical Logging to Extend Recovery to New Domains. *SIGMOD 1999* 73–84.

[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 TODS*, 17(1):94–162, March 1992.