High Speed On-line Backup When Using Logical Log Operations

David B. Lomet
Microsoft Research
One Microsoft Way, Redmond, WA 98052
email: lomet@microsoft.com

Abstract

Media recovery protects a database from failures of the stable medium by maintaining an extra copy of the database, called the backup, and a media recovery log. When a failure occurs, the database is "restored" from the backup, and the media recovery log is used to roll forward the database to the desired time, usually the current time. Backup must be both fast and "on-line", i.e. concurrent with on-going update activity. Conventional on-line backup sequentially copies from the stable database, almost independent of the database cache manager, but requires page-oriented log operations. But results of logical operations must be flushed to a stable database (a backup is a stable database) in a constrained order to guarantee recovery. This order is not naturally achieved for the backup by a cache manager concerned only with crash recovery. We describe a **"full speed"** backup, only loosely coupled to the cache manager, and hence similar to current on-line backups, but effective for general logical log operations. This requires additional logging of cached objects to guarantee media recoverability. We then show how logging can be greatly reduced when log operations have a constrained form which nonetheless provides very useful additional logging efficiency for database systems.

1 Introduction

Crash recovery requires that the stable database S be accessible and correct. Media recovery provides recovery from failures involving data in S. It is also a last resort to cope with erroneous applications that have corrupted S. To guard against stable database failures, the media recovery system provides (i) an additional copy of the database (called a backup B) and (ii) a media recovery log that is applied to the backup to roll its state forward to the desired state, usually the most recent committed state. To recover from failures, the media recovery system first *restores* S by copying B, perhaps stored on tertiary storage, to the usual secondary storage that contains S. Then the media recovery log operations are applied to the restored S to "roll forward" the state to the time of the last committed transaction (or to some designated earlier time).

Backing up the stable database is on-line if it is concurrent with normal database activity, "off-line" if concurrent activity is precluded. High availability requires on-line backup. Hence on-line backup is our focus, and in particular, on-line backup when logical operations, those that involve more than a single object or page, are logged. Other elements of media recovery do not necessarily require new techniques.

- Restoring the erroneous (part of) S with a copy from B is usually done off-line because media failure frequently precludes database activity. Off-line restore has little impact on availability as it occurs after media failure, a low frequency event. Off-line restore poses no technical problems unique to logical operations.
- Maintaining the media recovery log is conventional and is not impacted by the choice of log operations.
- Rolling forward the restored S involves redo recovery, which, for logical operations, has been described in our earlier work [10, 11].

1.1 Log Operations

Traditional Forms

Traditionally, database systems exploit two kinds of log operations.

Physical: A physical operation updates exactly one database object. No objects are read. Data values to be used in the update come from the log record itself. An example of this is a physical page write, where the value of the target page is to be set to a value stored in the log record.

Physiological: A physiological operation [4] also updates a single object, but also reads it. Hence, a physiological operation denotes a change in the object's value (a state transition). This avoids the need to store the entire new value for a target object in the log record. An example is

MOD 2000, Dallas, TX USA

the insert of a record onto a page. The page is read, the new record (whose value is stored in the log record) is inserted, and the result is written back to the page.

These two forms of log operations (also called page-oriented operations) make cache management particularly simple. Updated (dirty) objects in the cache can be flushed to S in any order, so long as the write ahead log (WAL) protocol is obeyed. For databases, pages are the recoverable objects and records are frequently the unit of update. Both are small. Thus, the importance of simple cache management can be allowed to control the form of log operation, restricting operations to the traditional varieties.

Logical Operations

When extending recovery to new domains, the cost of logging becomes a major consideration. Logical log operations can greatly reduce the amount of data written to the log, and hence reduce the normal execution cost of providing recovery. A log operation is logical [11], as opposed to page-oriented, if the operation can read one or more objects (pages) and write (potentially different) multiple objects.

Some examples of how logical logging can substantially reduce the logging required during normal execution are:

Application Recovery: Logical log operations for recovering application A's state [8] are:

$R(X, A)$**:** A reads X into its input buffer, transforming its state to a new state A'. Unlike page-oriented operations, the values of X and A' are not logged.

$W_l(A, X)$**:** A writes X from its output buffer. A's state is unchanged. Unlike page-oriented operations, we do not log the new value of X.

$Ex(A)$**:** The execution of A between resource manager calls is a physiological operation that reads and writes A's state. Execution begins when control is returned to A, and results in the new state when A next calls the resource manager.

File System Recovery: Logical operations can reduce logging cost for file system recovery. A copy operation copies file X to file Y. This same operation form describes a sort, where X is the unsorted input and Y is the sorted output. With logical operations, only source and target file identifiers are logged. With page oriented operations, one can't avoid logging the value of Y (or X).

Database Recovery: Logical operations are useful for, e.g., B-tree splits. A split operation moves index entries with keys greater than the split key from the old page to the new page. A logical split operation avoids logging the initial contents of the new page, which is unavoidable when using page-oriented operations.

The key to the logging economy of logical operations is that we can log operand identifiers instead of operand data values because the data values can come from many objects in the stable state. Since operand values can be large (applications or files may be several megabytes), logging an identifier (unlikely to be larger than 16 bytes) is a great saving.

Logical operations complicate cache management because cached objects can have flush order dependencies. As an example, for the operation *copy(X,Y)*, copying the value of object X to the object Y, we must ensure that the updated value of Y is flushed to S before we permit object X (if it has been subsequently updated) to be flushed to S, overwriting its old value. If an updated X is flushed before Y is flushed, a system failure will lose the old value of X needed to make replay of the copy operation possible. Hence, a subsequent redo of the copy operation will not produce the correct value for Y. These flush dependencies complicate the task of high speed on-line backup.

1.2 Existing Database Backup Techniques

In early database systems, the database was taken off-line while a backup was taken. This permitted a transaction or operation consistent view of the database to be copied at high speed from the "stable" medium of the database. Such off-line techniques work for all log-based recovery schemes and permit high speed backup. But the system is then unavailable during the backup process. Current availability requirements usually preclude this approach.

Backup is mentioned in [2, 5, 1, 4, 12, 13]. Gray [2] has the earliest discussion of "fuzzy dumps" and how to optimize media recovery by preprocessing the media recovery log. Haerder and Reuter [5] describe "archive recovery" but do not provide detail on how the on-line "fuzzy dump" is actually created or used. Bernstein et al [1] note that "The techniques used to cope with media failures are conceptually similar to those used to cope with system failures." Gray and Reuter [4] describe the "fuzzy dump" by reference to "fuzzy checkpoints", as used with system failures. The ARIES paper [12] describes this as "fuzzy image copying (archive dumping)". The high speed of these backup techniques depends upon constructing the backup by copying directly from the stable database S to the backup database B, independent of the cache manager.

There are other, incremental approaches to the backup problem. We have previously suggested [9] that a temporal index structure can be managed to ensure there is adequate redundancy so that recovery can restore the current state. But this non-conventional approach cannot currently be exploited because database systems lack temporal index support. Mohan and Narang [13] describe a method that works in a more conventional setting, and we return to this topic in the last section of the paper.

Our interest here is in conventional on-line database backup that copies data at high speed from the active S

to the backup B while update activity continues. Hence the state captured in B is fuzzy with respect to transaction boundaries. Coordination between backup process and active updating when traditional log operations are used occurs at the disk arm. That is, backup captures the state of an object either before or after some disk page write (we assume I/O page atomicity). B remains recoverable because page-oriented operations permit the flushing of pages to a stable database in any order. Since logged operations are all page-oriented, B is (i) *operation consistent*, i.e., results of an operation are either entirely in B, or are entirely absent; and (ii) selective redo of logged operations whose results are absent from B will recover the current database S.

The media recovery log must, of course, include all operations needed to bring objects up-to-date. The on-line system, which is updating S and logging the update operations, does not know precisely when an object is copied to B, and so needs to be "synchronized" with the backup process to make sure the log will contain the needed operations. For page-oriented log operations, synchronization between backup and cache manager only occurs at the begining of the backup. (Data contention during backup to read or write pages is resolved by disk access order.) The media recovery log scan start point can be the crash recovery log scan start point at the time backup begins. B will include all operation results currently in S at this point, plus some that are posted during the backup. Hence, this log, as subsequently updated by later updates, can provide recovery to the current state from B as well as from S. Subsequently, backup is independent of the cache manager, and can exploit any technique that it wishes to effect a high speed copy. This usually involves sweeping through S copying pages in a convenient order, e.g., based on physical location of the data. Different parts can be copied in parallel as well.

1.3 The Problem

Existing database backup methods do not work with logical operations and cannot support an on-line backup involving high speed copying while update activity continues. The fuzzy backup technique described above depends on logged operations being page-oriented. But logical log operations [10, 11] can involve multiple pages (or objects) and updated objects (e.g. pages) must be flushed to S in a careful order for S to remain recoverable. **Objects updated by logical operations have the same ordering constraints when "flushed" (copied) to the backup database B to ensure correct media recovery.**

The fundamental problem here is that we have two databases, S and B, upon which flush dependencies must be enforced. A "logical" solution to this problem is to stage all copying from S to B through the cache manager, and flush dirty data synchronously (a "linked" flush) to both S and B. (That is, dirty data flushed to S is also flushed to B such that the next flush of dirty data does not commence until the prior "linked" flush to both S and B has completed.) Of course this is a completely unrealistic solution. Copying from S to B via the database cache is unrealistic for page-oriented operations because of the performance impact. Pursuing this for logical log operations, where "linked" flushes are required is even less realistic.

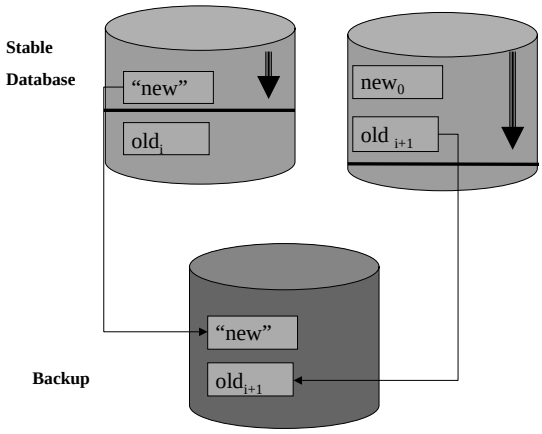


Figure 1: A B-tree backup problem arises for the sequence: backup("new") to B; flush(new_0) to S; flush(old_{i+1}) to S; backup(old_{i+1} to B). Backup B has the new version old_{i+1} of old, but not new_0 for new, even though new_0 was flushed to S prior to old_{i+1}
.

To efficiently create an on-line backup requires an "asynchronous" copy process that does not "go through the cache manager". But the task of keeping B recoverable so that media failure recovery is possible is the same task as with crash recovery and the stable database S. We must constrain flushing so that the flush dependencies we enforce for S are also enforced for B. **Unfortunately, when an asynchronous on-line backup is in progress, flush dependencies enforced for S can be violated for B.** And because of that, the fuzzy dumps of section 1.2 are not guaranteed to be recoverable.

The B-tree split example of Figure 1 illustrates the problem. The old node is updated and half of its contents moved to a new node. A logical split $MovRec(old, key, new)$ identifies the source old (in state old_i) and the target new (updated to state new_0). The log operation does not include the data moved. A second operation $RmvRec(old, key)$ deletes the copied records from old, producing state old_{i+1}. Should the $MovRec$ operation need to be replayed during recovery because new is not in state new_0 in S, old must be available in its pre-split state old_i containing the copied records. Hence, our write graph requires that new be flushed to S prior to old being overwritten with old_{i+1}, which lacks the copied records. This sequence ensures that S is recoverable should the system crash.

Consider now that a backup is in progress, indicated by the large arrow in the stable database of Figure 1. The part of the database in which new resides has already been

copied to B by an in-progress backup before this flushing activity occurs, but the part in which old resides has not. At completion of the backup, old has updated value old_{i+1} in B, but new's updated new_0 value is not present. This violates the required flush order for B. Indeed, the records moved from old to new are nowhere in B. Since a logical operation $MovRec$ was used to log the split, its log record does not include these records either. Hence, B cannot be successfully recovered to the current state to support media recovery.

1.4 Our Contribution: Backup with Logical Ops

So the situation that confronts us is that the "linked flush" approach has unrealistic performance, and the traditional fuzzy dump of S to B doesn't work.

We show how to backup a database system that logs logical operations by taking a high-speed backup in a manner that is similar to but subtlely different from a fuzzy dump for a database using only page-oriented operations. Like the best of the prior fuzzy dump algorithms, the database cache manager is bypassed. Our method ensures that the results of logged logical operations are "flushed" to B in the correct order, and hence that B remains recoverable [10], without tight coupling with the cache manaer. We exploit three insights:

1. an object can be written to the log as a substitute for being flushed to S or B. The object version needed for media recovery is then available from the (media) log.

2. knowledge of the order in which objects in S are copied to B can avoid some of the extra logging described above, hence improving backup efficiency.

3. limiting the forms of logical operations can limit the flush ordering, and exploiting this can further reduce the logging required to keep the backup recoverable.

1.5 Organization of Paper

In section two, we describe the redo recovery framework [10] that defines the requirements for redo recovery: installation graph, write graphs, unexposed objects, redo test, etc. Section three describes how to provide high speed, on-line backup when general logical operations are logged. This incurs extra cost for writing extra data to the log, however. So in section four, we limit ourselves to "tree" operations [10], a restricted class of logical operation which is nonetheless more general than the class of page-oriented operations. They permit, for example, the logging of B-tree splits without logging any data records. The tree operation class greatly reduces the need for extra logging. In section five, we provide a simple analysis to quantify the extra logging required for both general and tree operations. The analysis almost surely overstates the amount of extra logging, but suggests that this cost is quite modest. We end with a brief discussion of issues and directions in section six.

Operations	
Ex(A)	Application Execute: reads and writes *A*
R(A,X)	Application Read: reads *A,X* and writes *A*
$W_P(X,log(v))$	Physical Write: writes *X* with *v from log*
$W_{PL}(X)$	Physiological Write: reads and writes *X*
$W_L(A,X)$	Application Logical Write: reads *A*, writes *X*
$W_{IP}(X,log(X))$	CM Identity Write of *X* with its current value
Operation	***Op* Attributes**
readset(Op)	Read set of operation *Op*
writeset(Op)	Write set of operation *Op*
Write Graph	**Node *n* Attributes**
ops(n)	Set of operations associated with node *n*
vars(n)	Subset of *Writes(n)* flushed to install *ops(n)*
Reads(n)	$\cup$ {*readset(Op)*| *Op* in *ops(n)*}
Writes(n)	$\cup$ {*writeset(Op)*| *Op* in *ops(n)*}
W	Write graph – intersecting writes
rW	Refined write graph – exploiting unexposed

Table 1: Introduced notation

2 Flush Dependencies and Recoverability

Here we describe informally the important elements of redo recovery [10]. We focus on how logical operations give rise to flush dependencies among cached objects, i.e., careful ordering of the flushes in order to keep the S recoverable. We introduce some notation, summarized in Table 1, to make the subsequent exposition more concise and precise.

2.1 Overview

There are three key elements of redo recovery:

an installation graph prescribes the order in which the effects of operations must be "placed" into S in order for recovery to be possible. An installed operation is one that needn't be replayed during recovery because its effects do not need to be regenerated. An uninstalled operation is one that needs to be replayed for recovery to succeed. It is sometimes possible to install operations without changing S.

a write graph translates installation order on operations to flush order on updated objects. If updated objects are flushed to S in write graph order, then their updating operations are installed into the S in installation order. We focus here on write graphs, which manifest the flush dependencies with which we need to deal.

a redo test that is used during recovery to determine which operations to replay. Clearly, all operations considered uninstalled by the cache manager during normal execution will require replay. Redo tests can be relatively crude, and result in the replay of additional operations, and recovery can still succeed.

If the cache manager flushes updated objects to S in write graph order, installation order is enforced and recovery, exploiting a redo test, will recover S.

2.2 Installation Graph

The installation graph resembles the conflict graph of serializability theory, but is weaker. Installation graph nodes are log operations and edges are conflicts between operations. A log operation Op reads objects $readset(Op)$ and writes $writeset(Op)$. There are two kinds of installation edges from operation O to P for $O < P$ in conflict order:

read-write: ($readset(O) \cap writeset(P) \neq \phi$) This order is violated when P's updates are installed in the stable database before O's updates. A crash immediately after P's updates are installed would require that O be replayed to produce the missing effects. But this would be impossible because $readset(O)$ has changed. Hence, the database will not be recoverable.

write-write: When recovery is LSN-based (the usual case, and what we assume here), database state is never reset (rolled back to an earlier state) during recovery, and hence write-write edges($writeset(O) \cap writeset(P) \neq \phi$) are usually implicitly enforced.

Write-read conflicts ($writeset(O) \cap readset(P) \neq \phi$) are not installation graph edges. Installing an operation B that reads X before an earlier operation A that wrote X does not impair our ability to replay A. Only subsequent updates of objects in A's read set makes A's replay impossible. Of course, during the normal execution, B will see A's updates, so B's updates will correctly serializes after A, even when A is not installed before B.

2.3 Explainable States, Recovery, and Redo Test

After a crash, we must identify a set I of installed operations that *explain* S. We require that I be a prefix of the installation graph, so that installation order is enforced. (A prefix I is a subset such that if P is in I, then any $O < P$ is also in I.) Then a prefix I of the installation graph *explains* S if for each object X in S, either:

exposed: X's value is needed for recovery. Then it equals the value written by the last operation (in conflict order) in I that wrote to X; or

unexposed: X will be overwritten during recovery before it is read by an uninstalled operation. We do not care what X's value is since recovery does not depend on it.

After a crash, S is recoverable if it is explainable and the log contains all uninstalled operations (those that need to be redone). The recovery task of cache management is to guarantee that there is always an I that explains S. A redo test, REDO, ensures that replayed operations keep S explainable. A fundamental result from [10] is that if I explains S and O is a minimal (in conflict graph order) uninstalled operation, then O is indeed applicable (it finds its inputs in the same state as during normal execution) and the resulting extension of I to $I \cup \{O\}$ explains the new state of S and hence permits recovery to continue.

2.4 Write Graphs

A cache manager (**CM**) divides volatile state (cache) into "dirty" part (objects whose cached version is not in S) and "clean" part (objects whose cached version is in S, hence already installed and not discussed here). The cache manager manages the cache contents by reading new objects into and purging ojbects from the cache. Flushing a dirty object to S, thus installing its update operations, is one way to enable an object to be purged from the cache.

The CM must ensure that there is at least one I that explains the stable state S to ensure that S remains recoverable. It exploits a write graph WG for this [10]. Each WG node v represents a set $ops(v)$ of uninstalled operations and a set $vars(v)$ of the objects (variables) these operations write. There is an edge from node v to node w in WG if there is an installation graph edge from an operation P in $ops(v)$ to an operation Q in $ops(w)$. Operations of $ops(v)$ are installed by flushing the last values written to the objects of $vars(v)$ (making them a part of S). The objects of $vars(v)$ are flushed together atomically to guarantee operation atomicity. The $vars(v)$ sets must be flushed in write graph order to enforce installation order.

Many write graphs can be derived from an installation graph. A single node WG that calls for atomically flushing the entire cache is always sufficient to ensure recovery regardless of installation graph. However, the more nodes into which the uninstalled operations and the objects that they write can be divided, the more flexibility the cache manager has. Page-oriented operations can have degenerate write graphs, each node v having $|vars(v)| = 1$, and with no edges between nodes and hence no restrictions on flush order. Not so for logical operations. Logical operations can result both in nodes where $|vars(v)| > 1$ and with a required flush order between nodes.

The "Intersecting Writes" Write Graph

The "intersecting-writes" write graph W of [10] usually has several nodes. The algorithm for generating W uses the idea of collapsing a graph A with respect to a partition P of its nodes. The result is a graph B where each node w corresponds to a class in the partition P. An edge exists between nodes v and w of B if there is an edge between nodes a and b of A contained respectively in v and w. There are two "collapses". During the first collapse, each partition class of P represents nodes corresponding to operations with write sets that intersect. In the second collapse, which makes the graph acyclic and hence a feasible flush order, each class of P denotes a strongly connected region of the graph.

In addition to the problem of flush dependencies, a write graph can require that multiple objects be flushed together

atomically. Multi-object flush sets arise in two ways:

- op O updates several objects ($|writeset(O)| > 1$) or
- when cycles are removed during the second collapse of strongly connected regions R to node m, requiring the atomic flushing of $vars(m) = \cup\{vars(n)|n \in R\}$.

There is no way to remove objects from $vars(n)$ for any node n of W. $|vars(n)|$ increases monotonically, resulting in ever larger atomic flushes, until $vars(n)$ is finally flushed. This is highly unsatisfactory.

A Write Graph Exploiting Unexposed Objects

We need a WG that permits objects to be removed from $vars(n)$ of a node n when appropriate operations occur. This can be attained by exploiting objects that are unexposed. We alter the initial collapse of W above to construct a new "refined" write graph rW [11]. (We still need the second collapse to make rW acyclic.) The result is a write graph with more nodes n and fewer objects in $vars(n)$.

Only exposed objects need values from the last(in conflict order) installed operations that write them. Because of this, a "blind write" of X, which does not read the prior values of X, can remove X from $vars(n)$ of an existing node. The simplest blind write is a physical write $W_P(X, log(v))$ that reads no objects in the stable database (the value v comes from the log record) and writes X. Because a blind write does not require the prior value of X, this value is not needed for future recovery of X itself. To ensure that v is not needed for recovery, all operations that read X's v are installed prior to installing node n. We introduce an extra write graph edge from a node m, with an operation in $ops(m)$ that reads X's v, to node n (called an inverse write-read edge) that causes m to be purged from the cache before n. No uninstalled operation's recovery then depends upon the old value of X when n has no write graph predecessors and can be purged (via flushing $vars(n)$). X has become unexposed and we do not need to flush it. Hence we can remove X from $vars(n)$.

There are two salient differences between W and rW:

- In W, $vars(n) = Writes(n)$. In rW, not all objects in $Writes(n)$ are in $vars(n)$. Nonetheless, we install all operations in $ops(n)$ by flushing only $vars(n)$.
- There may be extra edges between nodes n and m in rW to ensure that certain objects are not exposed, and these need not be installation graph edges between operations in $ops(n)$ and $ops(m)$.

The example of Figure 2 illustrates how a blind write operation removes objects from a multi-object atomic flush set in rW. Multiple objects written by operation A are in node 1's atomic flush set, i.e., $vars(1) = \{X, Y\}$. After operation C, $vars(1) = \{Y\}$.

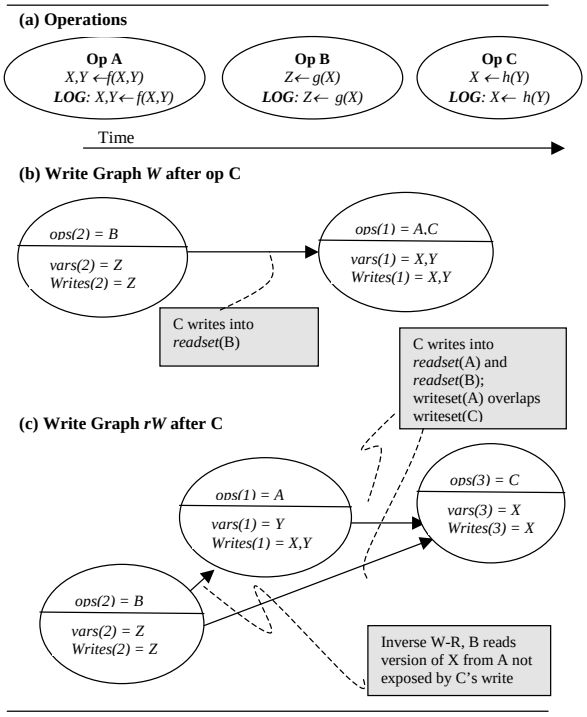


Figure 2: Write graphs rW and W when an X becomes unexposed. W has one node for X and Y, requiring their atomic flushing. rW has separate nodes for X and Y, the unexposed X being removed from $vars(1)$.

2.5 Cache Manager Identity Writes

rW only **makes possible** the shrinkage of flush sets $vars(n)$. To remove objects from a $vars(n)$ requires an operation of the appropriate form, and such an operation might not arise. However, the cache manager can itself remove objects from $vars(n)$ without atomically flushing them. It initiates an identity write operation $W_{IP}(X)$ that "writes" an object of $vars(n)$ without changing it and that is logged as a physical operation by writing the value of X to the log. This operation results in a new node m with $ops(m) = \{W_{IP}(X)\}$ and $vars(m) = X$. Importantly, it removes X from $vars(n)$. When n has no predecessors, $vars(n)$ (now not including X) can be atomically flushed, installing all operations of $ops(n)$. This technique works with arbitrary log operations. Later values for unexposed objects removed from $vars(n)$, which are not flushed, are recovered via the values logged for them as a result of their identity writes.

We have resorted here to logging physical writes to effectively manage the cache when $|vars(n)| > 1$. This approach was shown in [11] to be lower cost and have less impact on system operation than atomic flush transactions. Further, multiple updates can accumulate in each object before we log or flush it. Hence, as is common in database systems, the cost of flushing (and logging) is amortised over several updating operations, a substantial saving.

3 Backup for General Logical Operations

In this section, we describe how to cope with general logical operations that can read and write multiple objects, while supporting on-line creation of backup database B. The basic task is to install operations in B in installation order, by flushing dirty cached objects in write graph order, and hence to ensure that B remains recoverable.

3.1 Backup Overview

Preserving backup B's recoverability is difficult because we do not know when an object will be copied to B when the backup is only loosely synchronized with cache management, as illustrated in Figure 1. We need to enforce the flush order prescribed by the write graph for the flush to B when we flush an object to S. But that write graph may be different from the write graph that exists when an object is copied to B. For general operations, where it can be impossible to anticipate what a later write graph may be, we need to take the conservative approach of guaranteeing installation order regardless of changes to the write graph.

There are two aspects to our approach when logged operations are logical and hence create flush order dependencies.

Backup Progress We define a backup order (total or partial) that constrains when objects in S will be copied to B. Each recoverable object is in this ordering. We track the progress of backup in terms of how far it has gone in this ordering. While this constrains the backup process, backup retains great flexibility in how it sequences the copying of S to B. We characterize the backup process in terms of whether a backup is active and its progress in terms of the backup order. The testing of this characterization represents the synchronization required between the backup process and cache management. This is similar to the conventional fuzzy dump process, but we need added synchronization to reduce backup costs.

Cache Management The backup progress relative to the objects that we wish to flush from the cache determines how we manage the cache. Either we can flush objects from the cache normally to install operations, or we inject cache manager identity writes that ensure that object values are present on the log when they are needed and might not end up in B. This is installation without flushing (Iw/oF)into B, which we describe in the next subsection.

3.2 Installing Without Flushing

We saw in section 2.5 how cache manager identity writes could be used to avoid having to flush multiple objects to the stable database atomically. We can, when attempting to effect the installation of operations in a backup, reduce the number of objects requiring flushing (i.e. in $vars(n)$) to zero. We call this *"Installing without Flushing"* or **Iw/oF**. Every object in $vars(n)$ for a write graph node n whose $ops(n)$ we wish to install is updated via a cache manager identity write. Each of these operations removes its target object from $vars(n)$ and writes its value to the log. At the end, there are no objects in $vars(n)$. When n has no write graph predecessor node and $vars(n)$ is empty, $ops(n)$ is installed in the stable database, without any of the objects in $vars(n)$ having been flushed.

What has happened is that the logged values for objects in $vars(n)$ permit us to recover the stable database without having actually written these values to the stable database, in our case, the backup. Logging the identity operations is just as effective for recovery as flushing because these log records permit the truncation of the log in the same way that flushing does. This log truncation was described in [11] where we showed how this permitted us to advance the $rLSN$ of each object so written. Intuitively, this ensures that objects not captured in the backup are captured in the media recovery log and hence are available for media recovery.

3.3 Dealing with Write Graph Nodes

We need to understand when a backup is under way and the state of its progress. We then know how to deal with write graph nodes. Assume write graph node n has no predecessor write graph nodes. To keep S recoverable from a system crash, we atomically flush the objects in $vars(n)$ to S. If backup is not in progress, this suffices. If backup is in progress, we also want to simultaneously install $ops(n)$ into B. (This permits us to rely on only a single write graph.) There are two cases.

$Pending:$ The objects of $vars(n)$ (usually, there will only be a single object so that atomic flushing can be ensured by disk write atomicity) are known to have not yet been backed up. Hence, their new values will become part of B if we write them to S. Therefore, this flush effectively installs (eventually) the operations of $ops(n)$ in B. Write graph nodes that must follow the current node to maintain installation order, in fact do so correctly for B as well as for S.

$\neg Pending:$ An object of $vars(n)$ *may* already have been copied to B. Hence flushing $vars(n)$ now to S does not necessarily install $ops(n)$ in B. However, we can install $ops(n)$ into B without flushing any objects. We introduce cache manager identity write log operations (W_{IP}) for $vars(n)$, as we have described above. These log operations are in the media recovery log for B because we are writing them to the log after backup has begun. (Of course, we can flush pending objects to S, and log only the non-pending objects.)

In both cases above, we have installed the operations $ops(n)$ of write graph node n into the active backup at the "same time" that we install them into the stable database. What we have not described is how we record the progress of the backup, how we use backup order to determine whether objects have already been backed up, and exactly how the cache manager coordinates with the backup process

to guarantee that the assumptions we are exploiting are valid during the cache manager process of handling write graph nodes. We consider these aspects below. We then describe a specific cache management algorithm.

3.4 Backup Progress

We need to discuss two things in this subsection, the backup ordering, and how backup progress is tracked relative to this order. It is this ordering that tells us whether an object being flushed to S will appear in B.

Backup Order

With each object X, we associate a value $\#X$ in the backup [partial] order such that for any other object $\#Y$, if $\#X < \#Y$, then X is guaranteed to be copied to B before Y. Where these values are not ordered, no knowledge exists about the relative order of copying the objects to B. We track the progress of a backup in terms of these values, which can be derived from the physical locations of data on disk.

It is possible to divide the database into disjoint partitions, and to independently track backup progress in each partition. This permits us to back up partitions in parallel. If no single operation can read or write objects from more than a single partition, we can independently track backup progress for each partition and hence arbitrarily interleave the copying of the partitions from S to B. (The degenerate case where each object is a partition is produced when operations are page-oriented, and backing up a partition is an atomic action.)

Tracking Backup Progress

To permit backup to proceed with little synchronization between it and the cache manager, we deliberately introduce some fuzziness in how we track its progress. Thus backup reports its progress only from time to time. Depending on system tuning considerations, the reporting can be made more or less precise. We control this by varying the granularity of the steps in which we report backup progress.

For each partition in which backup progress is to be tracked independently (there may be one or several), we maintain two values, D (Done) and P (Pending), in the backup order, that divides objects in the partition into three sets, as described below and as shown in Figure 3.

$Done(X)$: When $\#X < D$, X has been copied to B. Hence, X, if flushed now, will not appear in B.

$Pend(X)$; When $\#X > P$, X has not yet been copied to B. Hence, X, if flushed now, will appear in B.

$Doubt(X)$: When $D < \#X < P$, we do not know whether or not X has been copied to B. Hence, if X is flushed now, we do not know whether it will appear in B or not.

If $\#X$ is not ordered relative to P or D, then we do not know whether it has been backed up or not, and hence treat it as if it were in doubt ($Doubt$).

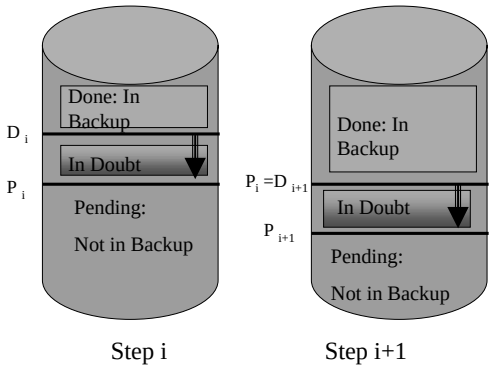


Figure 3: **Tracking backup progress.** At each step, the previously "in-doubt" part of S becomes part of S known to be copied to B. The "pending" part, previously known to not be in B, is partitioned into "in doubt" and still "pending". The values of P and D track this.

We require that there be maximum value Max and minimum value Min such that for all X, $Min < \#X < Max$. These permit us to initiate, terminate, and track the progress of the backup. Between backups, we set $D = P = Min$. This tells us that no object has been copied to a current backup B, and that all objects are pending for B, whenever it starts. When backup starts, we set P to some higher value P_1, leaving $D = Min$. Objects X with $\#X < P$ are in doubt as we do not track progress between $D = Min$ and $P = P_1$, the boundary for the first step. When backup has copied all objects less than P, it sets $D = P$, indicating that all the objects below P (i.e. in the first step) are done. It then sets $P = P_2$, the boundary for the second step. Now, only objects above P_2 are pending and hence guaranteed not yet to be in the current backup. Objects between $D = P_1$ and $P = P_2$ are in doubt.

Backup completes when P is set to Max, the highest value possible. This is the last step. At that point, there are no longer any pending objects and hence no objects, if now flushed, that are guaranteed to have their values for those objects appear in B. When backup completes this step, it resets to $D = P = Min$.

Our technique for charting backup progress reduces to a one step process when the only steps of backup progress are $D = P = Min$ and $D = Min, P = Max$. The only information then is whether backup is in progress or not. Backup need only synchronize with the CM when it changes the values of D and P. Hence, we can vary the granularity of synchronization (see below) from twice per backup (delimiting the duration of the backup) to many times, depending on the urgency to reduce the additional logging activity required to keep B recoverable.

Synchronization

While the CM is flushing objects to the stable database, it must know the state of backup progress. So it checks D

and P prior to flushing objects. However, unless additional measures are taken, the values for D and P can change while flushing is in progress. To prevent this requires synchronization between CM and backup process. This synchronization can be made at a granularity (in terms of number of steps) that is deemed appropriate for backup performance, but it is essential for correctness.

Since backup progress in each database partition is tracked independently, we define a backup latch per partition. This permits us to back up partitions in parallel. (Alternatively, we could back up the partitions sequentially, thus ordering the partitions, i.e. resulting in one large partition.) The backup latch protects D and P for a partition, hence synchronizing backup with the flushing activity of the cache manager.

When the backup process updates its progress, it requests the partition backup latch in exclusive mode. When granted, backup can update D and P. When updating is complete, backup releases the backup latch. When the cache manager flushes objects in $vars(n)$ of write graph node n, it requests the backup latch in share mode. This prevents backup from changing D and P while the CM is flushing objects. The CM then compares $\#X$ in $vars(n)$ with D and P, safe in the knowledge that these quantities will not change until it has completed its flushing activity and releases the backup latch for the partition. Share mode enables a multi-threaded CM to flush objects concurrently.

3.5 Cache Management Algorithm

Putting the above together yields the following algorithm. First, request the backup latch in share mode for the partition containing X in $vars(n)$, where n has no write graph predecessors. (We assume here that X is the only object in $vars(n)$, to enforce operation atomicity.) When granted, proceed, based on backup progress, as follows:

$Done(X)$**:** Install $ops(n)$ in B and S via Iw/oF. The value needed for X for media recovery will be on the media recovery log. We flush X to S before X is dropped from the cache in order to have the latest value of X available to us. Thus we both log and flush X before dropping it from the cache.

$Doubt(X)$**:** We proceed as for $Done(X)$. This may be unnecessary, but we cannot determine this, and this guarantees that B is recoverable.

$Pend(X)$**:** Merely flush X to S to install $ops(n)$ not only in the S, but also in B. The value for X needed for media recovery will be in B.

When the CM has completed flushing objects, it releases the backup latch.

4 Backup for Tree Operations

4.1 Description

In this section, we explore how to perform backup for a constrained set of log operations called "tree" operations [10]. These are more flexible than page-oriented operations and require less logging of data values. Like page-oriented operations, a tree operation can read and write an existing object, but it can also write an object not previously updated (a **new** object) in the same operation. Here, we modify this definition slightly. We include page-oriented operations ("physiological" operations denoted by W_{PL}) in the tree operation definition. But we do not permit an operation to update multiple objects. Instead, a "logical" write operation (W_L) writing the new object cannot update the old (read) object. Hence tree operations are of two forms:

1. Page-oriented: Possibly read an existing object old and write old, i.e. $W_{PL}(old)$ or $W_P(old, log(v))$
2. Write-new: Read an existing object old and write a new object new, i.e. $W_L(old, new)$

Tree operations are useful as we can initialize a new object without logging its initial value. For example, B-tree node split operations can be very efficiently logged compared to page-oriented operations.

Page-oriented operations: The update of old is logged as $RmvRec(old, key)$ that removes records with keys larger than key from old. The update of new is logged as a physical write with records needed to initialize new in the log record, i.e. as $W_P(new, log(value_{new}))$.

Tree operations: We log the update of new, moving the high records from old to new, as $MovRec(old, key, new)$, The update of old is logged as before. $MovRec$ must precede $RmvRec$ because the updated old will not contain the moved records.

For tree operation write graphs, the only edges are between node n where $vars(n) = \{new\}$ and m with $vars(m) = \{old\}$ for which an operation reads old and writes new. This can produce a "branching" of the write graph. However, no joining of write graph paths, and no cycles requiring a collapse are introduced. This results in a write graph where each node n has $|vars(n)| = 1$. Hence, as with page-oriented operations, we can identify a write graph node with its updated object, e.g. new can identify both an object and the write graph node n with $vars(n) = \{new\}$. Thus, new is a predecessor of old (and old is a successor of new) in WG.

Successors for any object X in the cache are on at most one path of write graph nodes. A successor node may, however, have more than one predecessor as a single old object can spawn edges to multiple new objects. Tree operations are so-named because their write graph (when only tree operations are logged) is a set of trees.

We call old a potential successor of new if there has been an operation that reads old and writes new but old has not yet been updated, and we denote the union of successors and potential successors of a cached object X by $S(X)$. With only tree operations, $S(X)$ never increases for any X in the cache. $S(X)$ is fixed at the time that an object is first updated in the cache. Subsequent operations may add predecessors, but not successors because an object can only be a new object the first time it is updated.

4.2 Backup

We identify when to use Iw/oF to install operations of $ops(n)$ into B. Recall that the set $S(X)$ for each X in $vars(n)$ is fixed when n is added to the write graph. The properties of $S(X)$ and n together help determine how to install the operations in $ops(n)$. The most important factor in whether Iw/oF logging is needed when we flush X is how backup progress compares with the position of X and its successors, $S(X)$. To help us with that, we maintain $MAX(X) = max(\{\#y | y \in S(X)\}$ with each cached object X. When an operation $W_L(Y, X)$, reading Y and writing X, appears, we incrementally compute for the new object X, $MAX(X) = max(\#Y, MAX(Y))$. ($MAX(Y) = 0$ if Y has no successors.) Then we say that $S(X)$ is "done", "pending", or "in doubt" directly based on that state of $MAX(X)$. If $Max(X) \leq D$, then $Done(S(X))$ and no successor y of X will appear in B. Otherwise, when $Max(X) > D$, then $\neg Done(S(X))$, and some successor y of X might appear in B.

Our case analysis for when we can avoid Iw/oF logging depends on the state of backup progress. The cases are graphically shown in Figure 4.

$Pend(X) or Done(S(X))$**:** Either X will definitely appear in B ($Pend(X)$); or no y in $S(X)$ will be included in B (i.e. $Done(S(X))$). We can flush X without extra logging because flush order cannot be violated.

$(Done(X) \& \neg Done(S(X)))$ or $(\neg Pend(X) \& Pend(S(X)))$**:** We know that either X will not get to B and are unsure about $S(X)$ or $S(X)$ will get to B but we are unsure about X. Hence, to guarantee that X gets to B should any of $S(X)$ get to B, we use Iw/oF and write X to the log.

$Doubt(X) \& Doubt(S(X))$**:** When X and $S(X)$ are both in the region where backup is active, we are "in doubt" as to whether they will get to B. In this region, we exploit the † property which holds about half the time.

† **if** (any y in $S(X)$ gets to B when flushed to S) **then** (any earlier flush of X to S gets to B)

When † holds, flush order to the backup will not be compromised. If X does not get into B (which is the dangerous case), then neither will all of $S(X)$.

Consider first the case where $S(X) = \{y\}$, i.e. there is a single successor to X. X needs to be installed not later than

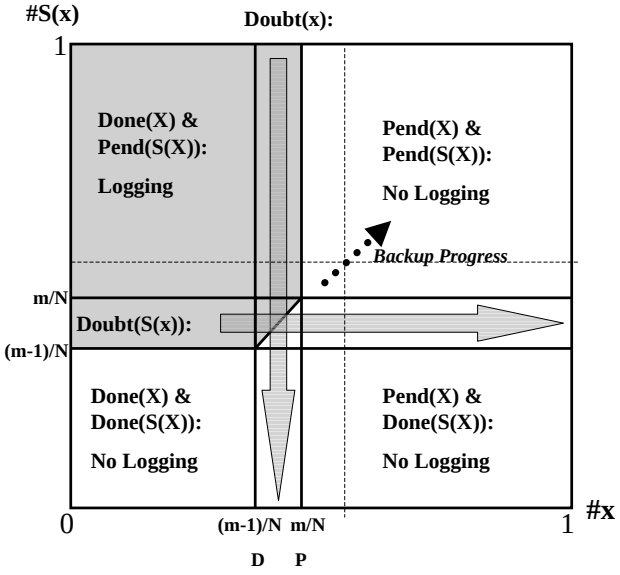


Figure 4: The regions of $< \#X, \#S(X) >$ space as to whether or not additional logging is required to ensure that flush order dependencies for S are enforced for B. The shaded area requires the extra Iw/oF logging.

y, i.e. if y is in the backup, X must also be in the backup. This is so when $\#y < \#X$.

When $|S(X)| > 1$, it is tempting to think that if $\#X > Max(X)$, then X can be flushed without extra logging. But this is not so. If there is an order violation between any pair of successors (i.e. † does not hold), then we must use Iw/oF for that successor. That ensures that there is a successor to X that is guaranteed to be installed in B. But we are unsure whether X will get to B. Hence we must use Iw/oF to ensure that X will also be installed in B. Thus, once an order violation appears among $S(X)$, any subsequently added predecessors of X also have an order violation and must likewise be installed in B using Iw/oF. Thus, each write graph node n has a $violation(n)$ flag. The $violation(X)$ flag is set if $\#X \leq \#y$ where y is an immediate successor resulting from an operation that read y and wrote X or if $violation(y)$ is set. This incrementally maintains $violation(X)$ for all X.

5 Logging Performance

In this section, we present a comparative analysis of how much extra logging is required by the cache manager when a backup is in progress. This extra logging is a function of how many steps there are in the backup- and hence of how frequently the backup process synchronizes with the cache manager to report its progress. We treat both general operations and tree operations, and quantify the added logging costs of going from physiological operations to tree operations to completely general operations.

We assume that a backup is done in N equal steps and

that we can clearly separate the database into N disjoint and approximately equal pieces. When we are at the mth step, the fraction of the database that has been definitely backed up (done) is $(m-1)/N$, and the fraction that is definitely not backed up yet ($Pend$) is $1 - m/N$, while the part of the database that is in the process of being backed up ($Doubt$) constitutes $1/N$ of the database. (These fractions sum to one, the entire database.) Hence, assuming that updated pages are uniformly distributed, we have:

- $Prob\{Done(X)\} = Prob\{\#X \leq D\} = \frac{m-1}{N}$
- $Prob\{Pend(X)\} = Prob\{\#X > P\} = 1 - \frac{m}{N}$
- $Prob\{Doubt(X)\} = \frac{1}{N}$

We determine analytically the probability that an object flush requires logging ($Prob_m\{log\}$) because Iw/oF is needed at backup step m, and then average over the N steps by summing the probability at each of the N steps and dividing by N. (There is no extra logging for media recovery when backup is not in progress.) This overall probability is

$$Prob\{log\} = \frac{1}{N}\sum_{m=1}^{N} Prob_m\{log\}$$

Now we need to determine $Prob_m\{log\}$.

5.1 General Logical Operations

For general operations, we must do extra logging for Iw/oF whenever we are not sure that the object X being flushed will be included in the current active backup. The probability of this is

$$Prob_m\{\neg Pend(X)\} = Prob\{\#X < P\} = \frac{m}{N}.$$

The cost averaged over all steps is

$$Prob\{log\} = \frac{1}{N}\sum_{m=1}^{N}\frac{m}{N} = \frac{1}{N^2}\sum_{m=1}^{N} m = \frac{1}{2}(1+\frac{1}{N})$$

For example, when $N = 1$, and we merely know when a backup is in progress, we must always do the extra logging. For high N, this cost approaches $\frac{1}{2}$.

5.2 Tree Operations

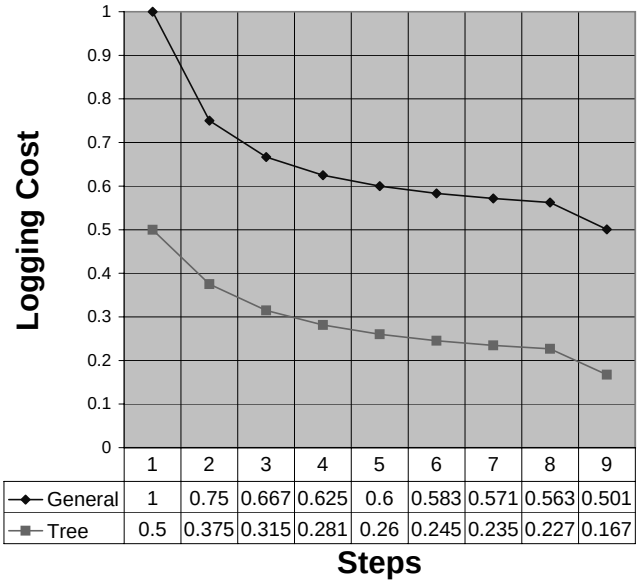
For tree operations, we assume that $|S(X)| = 1$, i.e. each cached object has exactly one successor. This is not realistic. First, an object might have no successors and be flushed without extra logging. (If X has one successor y, then y cannot have any successors.) Second, an object may have more than one successor. Nonetheless, this analysis provides some insight into the logging requirements. It surely overstates the logging cost relative to general log operations. For tree operations, we may find that an object has no successors, permitting us to frequently avoid any logging- while we get no such information for general operations, where successors can emerge at any time.

Subject to the preceding qualifications, the cost in extra logging at step m to flush an object X is

$Prob_m\{log\} = Prob\{\neg Pend(X) \& \neg Done(S(X)\}$
$- Prob\{Doubt(X) \& Doubt(S(X)) \& Prob\{\#S(X) < \#X\})$

Substituting prior values for $Pend$, $Done$, and $Doubt$, logging cost at step m becomes

$Prob_m\{log\} = \frac{m}{N}(1 - \frac{m-1}{N}) - \frac{1}{2N^2} = \frac{N+1}{N^2}m - \frac{1}{N^2}m^2 - \frac{1}{2N^2}$



	1	2	3	4	5	6	7	8	9
General	1	0.75	0.667	0.625	0.6	0.583	0.571	0.563	0.501
Tree	0.5	0.375	0.315	0.281	0.26	0.245	0.235	0.227	0.167

Figure 5: The frequency (probability) with which extra logging is required for general and tree operations as a function of the number of backup steps.

Averaging over all steps, we get
$Prob\{log\} = \frac{1}{N}\sum_{m=0}^{N}(\frac{N+1}{N^2}m - \frac{1}{N^2}m^2 - \frac{1}{2N^2}) = \frac{1}{6} + \frac{1}{2N} - \frac{1}{6N^2}$
Asymptotically, as the number of steps in the backup process increase, only one flush in six needs extra logging.

5.3 Assessing Results

Figure 5 contrasts the extra logging costs for general operations versus tree operations as the number of backup steps that are synchronized with the cache manager increases.

General operations require that an object be logged (Iw/oF) over half the time to flush it to the stable database. The more backup steps, the closer to 50% this is. If we only know that backup is active (one step), we must log all flushed variables.

Tree operations reduce logging (compared with general ops) by between half and two thirds, to about about one object flush in six. This fractional reduction increases as the number of backup steps increases, i.e the more steps, the fewer times that Iw/oF logging is needed.

In both cases, most of the reduction in logging (almost 90%) has been achieved with an eight step backup, so there is little incentive to further increase the number of backup steps.

Our analysis surely overstates the extra costs of supporting logical operations for high speed backups.

- Extra logging only ocurs during backup. Usually a database backup is only active a small part of the time, frequently during periods of low database activity. Hence, extra logging, when averaged over total time, is much less than what is reported here. Further, this extra

logging merely reduces somewhat the very substantial gain that comes from using logical operations.

- Extra logging can also substitute for flushing. Should X be dirty in the cache, but hot, such that we do not drop it from the cache, logging it to install its update operations in S treats S the way we have been treating B. To drop X from the cache requires a flush to S, but this may well be greatly delayed by further updates.

This demonstrates that a high performance, asynchronous, on-line backup is possible while logging logical operations.

6 Discussion

6.1 Incremental Backups

By identifying the portion of the database state S that has changed since the last backup, we need only back up that changed portion. This is called an "incremental backup". Many database systems support a form of this. A clever way to do it while supporting high speed full backup and avoiding the cache manager is described in [13]. A natural question is whether our approach lends itself also to such incremental backup. There are two aspects of incremental backup.

1. Identify the set of database objects updated since the last backup. Schemes that do this for page-oriented log operations can be adapted for our more powerful operations. Adaptation is required because recovery checkpointing and how the log truncation point is identified are different, at least in detail.

2. Copy the identified objects to the backup. This is essentially the same problem that we have for full backup. Its solution should be similar as well, involving Iw/oF logging and tracking backup progress.

Hence, much of the efficiency of [13] also holds for backup with logical log operations.

6.2 Application Read Operations and Backup

In [8], we introduced log operations that permit efficient logging for application recovery. The only logical log operation we pursued there was "application read"($R(X, A)$). This avoided physically logging the value of X or A, but required that A be flushed before a change to X was flushed. In all resulting write graphs, only applications are predecessors. If applications are the last objects included in a backup, we guarantee that the † property holds (if X is included in the backup when X is flushed, so is any earlier flush of A), and **no** Iw/oF **logging** is incurred for backup. This is yet another example of how constraining operations can increase efficiency.

6.3 Some Future Directions

Three areas for further research seem interesting to pursue.

1. Tree operations reduce the logged data, avoid cyclic dependencies, and update only a single page. Are there other more general operation classes that avoid substantial logging during a backup?

2. Media failure might affect only a small part of the database. With logical operations, it may not be easy to determine the database part upon which its recovery depends. Preventing operations from having operands from more than one partition makes a partition the unit of media recovery. Are there other simple techniques?

3. Media recovery can protect against some application errors that corrupt the database. In this case, we may not recover the latest database state, but a state that excludes the effects of the corrupting application. This is difficult now. Can we support this in a general way?

References

[1] Bernstein, P., Hadzilacos, V. Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison Wesley (1987).

[2] Gray, J. Notes on Data Base Operating Systems. IBM Tech Report RJ2188 (Feb. 1978), IBM Corp., San Jose, CA

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

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

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

[6] King, R. P., Halim, N., Garcia-Molina, H. Polyzois, C. A. Management of a remote backup copy for disaster recovery. ACM Trans. on Database Systems 16, 2 (June 1991) 338-368

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

[8] Lomet, D. Application recovery using generalized redo recovery. Intl. Conf. on Data Eng., Orlando (Feb. 1998) 154-163.

[9] Lomet, D. and Salzberg, B. Exploiting a History Database for Backup, VLDB Conference, Dublin (Sept. 1993) 380-390.

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

[11] Lomet, D. and Tuttle, M. Logical logging to extend recovery to new domains. ACM SIGMOD Conference, Philadelphia (May 1999) 73-84.

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

[13] Mohan, C. and Narang, I. An Efficient and Flexible Method for Archiving a Data Base. ACM SIGMOD Conference, Washington, DC (May 1993) 139-146.