Segment-Based Recovery: Write-ahead logging revisited

Russell Sears
UC Berkeley
sears@cs.berkeley.edu

Eric Brewer
UC Berkeley
brewer@cs.berkeley.edu

Although existing write-ahead logging algorithms scale to conventional database workloads, their communication and synchronization overheads limit their usefulness for modern applications and distributed systems. We revisit write-ahead logging with an eye toward finer-grained concurrency and an increased range of workloads, then remove two core assumptions: that pages are the unit of recovery and that timestamps (LSNs) should be stored on each page.

Recovering individual application-level objects (rather than pages) simplifies the handing of systems with object sizes that differ from the page size.

We show how to remove the need for LSNs on the page, which in turn enables DMA or zero-copy I/O for large objects, increases concurrency, and reduces communication between the application, buffer manager and log manager. Our experiments show that the looser coupling significantly reduces the impact of latency among the components. This makes the approach particularly applicable to large scale distributed systems, and enables a "cross pollination" of ideas from distributed systems and transactional storage.

However, these advantages come at a cost; segments are incompatible with physiological redo, preventing a number of important optimizations. We show how allocation enables (or prevents) mixing of ARIES pages (and physiological redo) with segments. We present an allocation policy that avoids undesirable interactions that complicate other combinations of ARIES and LSN-free pages, and then present a proof that both approaches and our combination are correct.

Many optimizations presented here were proposed in the past. However, we believe this is the first unified approach.

1. INTRODUCTION

Transactional recovery is at the core of most durable storage systems, such as databases, journaling filesystems, and a wide range of web services and other scalable storage architectures. Write-ahead logging algorithms from the database literature were traditionally optimized for small, concurrent, update-in-place transactions, and later extended for larger objects such as images and other file types.

Although many systems, such as filesystems and web services, require weaker semantics than relational databases, they still rely upon durability and atomicity for some information. For example, filesystems must ensure that metadata (e.g. inodes) are kept consistent, while web services must not corrupt account or billing information.

In practice, this forces them to provide recovery for some subset of the information they handle. Many such systems opt to use special purpose *ad hoc* approaches to logging and recovery. We argue that database-style recovery provides a conceptually cleaner approach than such approaches and that, with a few extensions, can more efficiently address a wide range of workloads and trade off between full ACID and weaker semantics.

Given these broader goals, and roughly twenty years of innovation, we revisit the core of write-ahead logging. We present *segment-based recovery*, a new approach that provides more flexibility and higher concurrency, enables distributed solutions, and that is simple to implement and reason about.

In particular, we revisit and reject two traditional assumptions about write-ahead logging:

- The disk page is the basic unit of recovery.
- Each page contains a *log-sequence number (LSN)*.

This pair of assumptions permeates write-ahead logging from at least 1984 onward [7], and is codified in ARIES [26] and in early books on recovery [2]. ARIES is essentially a mechanism for transactional pages: updates are tracked per page in the log, a timestamp (the LSN) is stored per page, and pages can be recovered independently.

However, applications work with variable-sized records or objects, and thus there may be multiple objects per page or multiple pages per object. Both kinds of mismatch introduce problems, which we cover in Section 3. Our original motivation was that having an LSN on each page prevents use of contiguous disk layouts for multi-page objects. This is incompatible with DMA (zero-copy I/O), and worsens as object sizes increase over time.

Presumably, writing a page to disk was once an atomic operation, but that time has long passed. Nonetheless, traditional recovery stores the LSN in the page so it can be atomically written with the data [2, 5]. Several mechanisms have been created to make this assumption true with modern disks [8, 31, 34] (Section 2.1), but disk block atomicity is now *enforced* rather than inherent and thus is not a reason *per se* to use pages as the unit of recovery.

VLDB '09, August 24-28, 2009, Lyon, France

We present an approach that is similar to ARIES, but that works at the granularity of application data. We refer to this unit of recovery as a *segment*: a set of bytes that may span page boundaries. We also present a generalization of segment-based recovery and ARIES that allows the two to coexist. Aligning segment boundaries with higher-level primitives simplifies concurrency and enables new optimizations, such as zero-copy I/O for large objects.

Our distinction between segments and pages is similar to that of computer architecture. Our segments differ from those in architecture in that we are using them as a mechanism for recovery rather than for protection. Pages remain useful both for space management and as the unit of transfer to and from disk. Pages and segments work well together (as in architecture), and in our case preserve compatibility with conventional page-oriented data structures such as B-trees.

Our second contribution is to show how to use segment-based recovery to eliminate the need for LSNs on pages. *LSN-free pages* facilitate multi-page objects and, by making page timestamps implicit, allow us to reorder updates to the same page and leverage higher-level concurrency.

However, segment-based redo is restricted to *blind writes*: operations that do not examine the pages they modify. Typically, blind writes either zero out a range or write an array of bytes at an offset. In contrast, ARIES redo examines the contents of on-disk pages and supports *physiological* redo. Physiological redo assumes that each page is internally consistent, and stores headers on each page. This allows the system to reorganize the page then write back the update without generating a log entry. This is especially important for B-trees, which frequently consolidate space within pages. Also, with carefully ordered page write back, physiological operations make it possible to rebalance B-tree nodes without logging updates.

Third, we present a simple proof that segment-oriented recovery and ARIES are correct. We document the trade offs between page- and segment-oriented recovery in greater detail and show how to build hybrid systems that migrate pages between the two techniques. The main challenge in the hybrid case is page reallocation. Surprisingly, allocators have long-plagued implementers of transactional storage.

Finally, segment-oriented recovery enables a number of novel distributed recovery architectures that are hindered by the tight coupling of components required by page-oriented recovery. The distributed variations are quite flexible and enable recovery to be a large-scale distributed service.

2. WRITE-AHEAD LOGGING

Recovery algorithms are often categorized as either *update-in-place* or based on *shadow copies*. Shadow copy mechanisms work by writing data to a new location, syncing it to disk and then atomically updating a pointer to point to the new location. This works reasonably well for large objects, but incurs a number of overheads due to fragmentation and disk seeks. Write-ahead logging provides update-in-place changes: a redo and/or undo log entry is written to the log before the update-in-place so that it can be redone or undone in case of a crash. Write-ahead logging is generally considered superior to shadow pages [4].

ARIES and other modern transactional storage algorithms provide *steal/no-force* recovery [15]. *No-force* means that the page need not be written back on commit, because a redo log entry can recreate the page during recovery should it get lost. This avoids random writes during commit. *Steal* means that the buffer manager can write out dirty pages, as long as there is a durable undo log entry that can recreate the overwritten data after an abort/crash. This allows the buffer manager to reclaim buffer space even from in-progress transactions. Together, they allow the buffer manager to write back pages before (steal) or after (no-force) commit as convenient. This approach has stood the test of time and underlies a wide range of commercial databases.

The primary disadvantage of *steal/no-force* is that it must log undo and redo information for each object that is updated. In ARIES' original context (relational databases) this was unimportant, but as disk sizes increased, large objects became increasingly common and most systems introduced support for *steal/force* updates for large objects. *Steal/force* avoids redo logging. If the write goes to newly allocated (empty) space, it also avoids undo logging. In some respect, such updates are simply shadow pages in disguise.

2.1 Atomic Page Writes?

Hard disks corrupt data in a number of different ways, each of which must be dealt with by storage algorithms. Although segment-based recovery is not a panacea, it has some advantages over page-based techniques. Errors such as catastrophic failures and reported read and write errors are *detectable*. Others are more subtle, but nonetheless need to be handled by storage algorithms. *Silent data corruption* occurs when a drive read does not match a drive write. In principle, checksumming in modern hardware prevents this from happening. In practice, marginal drive controllers and motherboards may flip bits before the checksum is computed, and drives occasionally write valid checksummed data to the wrong location. Checksummed page offsets often allow such errors to be detected [8]. However, since the drive exhibits arbitrary behavior in these circumstances, the only reliable repair technique, *media recovery*, is quite expensive, and starts with a backup *checkpoint* of the page. It then applies every relevant log entry that was generated after the checkpoint was created.

A second, more easily handled, set of problems occurs not because of *what* data the drive stores, but *when* that data reaches disk. If write caching is enabled, some operating systems (such as Linux) return from synchronous writes before data reaches the platter, violating the write-ahead invariant [28]. This can be addressed by disabling write caching, adding an uninterruptable power supply, or by using an operating system that provides synchronous writes.

However, even synchronous writes do not atomically update pages. Two solutions to this problem are *torn page detection* [31], which writes the LSN of the page on each sector and *doublewrite* buffering [34], which, in addition to the recovery log, maintains a second write-ahead log of all requests issued to the hard disk. Torn page detection has minimal log overhead, but relies on media recovery to repair the page, while doublewrite buffering avoids media recovery, but greatly increases the number of bytes logged. Doublewrite buffering also avoids issuing synchronous seek requests, giving the operating system and hard drive more freedom to schedule disk head movement.

Assuming sector writes are atomic, segment-based recovery's blind writes repair torn pages without resorting to media recovery or introducing additional logging overhead (beyond preventing the use of physiological logging).

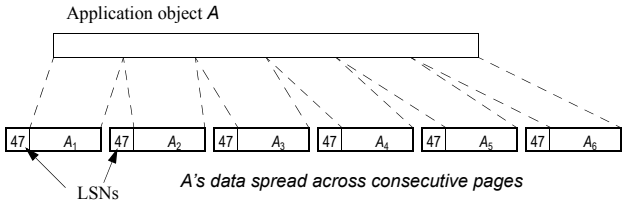

Figure 1: Per page LSNs break up large objects.

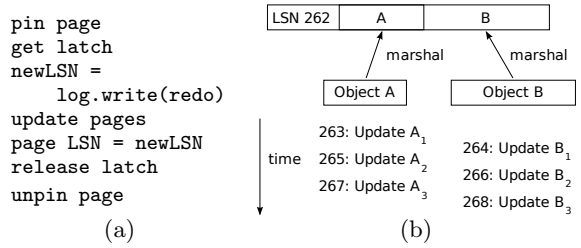

Figure 2: (a) Record update in ARIES. Pinning the page prevents the buffer manager from stealing it during the update, while the latch prevents races on the page LSN among independent updates. **(b)** A sequence of updates to two objects stored on the same page. With ARIES, A_1 is marshaled, then B_1, A_2 and so on. Segments avoid the page latch, and need only update the page once for each record.

3. PAGE-ORIENTED RECOVERY

In the next four subsections, we examine the fundamental constraints imposed by making pages the unit of recovery. A core invariant of page-oriented recovery is that each page is self-consistent and marked with an LSN. Recovery uses the LSN to ensure that each redo entry is applied exactly once.

3.1 Multi-page Objects

The most obvious limitation of page-oriented recovery is that it is awkward when the real record or object is larger than a page. Figure 1 shows a large object A broken up into six consecutive pages. Even though the pages are consecutive, the LSNs break up the continuity and require complex and expensive copying to reassemble the object on every read and spread it out on every write (analogous to *segmentation and reassembly* into packets in networking).

Segment-oriented recovery eschews per page LSNs, allowing it to store the object as a contiguous segment. This enables the use of DMA and zero-copy I/O, which have had significant impact in filesystems [9, 32].

3.2 Application/Buffer Interaction

Figure 2(a) shows the typical sequence for updating a single record on a page, which keeps the on-page version in sync with the log by updating them together atomically. In a traditional database, in which the page contains a record, this is not a problem; the in-memory version of the page is the natural place to keep the current version.

However, this creates problems when the in-memory page is not the natural place to keep the current version, such as when an application maintains its own working copies, and stores them in the database via either marshaling or an object-relational mapping [14, 16]. Other examples include BerkeleyDB [30], systems that treat relational databases as "key-value" storage [34], and systems that provide such primitives across many machines [6, 22].

Figure 2(b) shows two independent objects, A and B, that happen to share the same page. For each update, we would like to generate a log entry and update the object *without* having to serialize each update back onto the page. In theory, the log entries should be sufficient to roll forward the object from the page as is. However, with page-oriented recovery this will not work. Assume A has written the log entry for A_1 but has not yet updated the page. If B, which is completely independent, decides to then write the log entry for B_1 and update the page, the LSN will be that of B's entry. Since B_1 came after A_1, the LSN implies that the changes from A_1 are reflected in the page even though they are not, and recovery may fail.

In essence, the page LSN is imposing artificial ordering constraints between independent objects: updates from one object set the timestamp of the other. This is essentially *write through caching*: every update must be written all the way through to the page.

What we want is *write back caching*: updates affect only the cache copy and we need only write the page when we evict the object from the cache. One solution is to store a separate LSN with every object. However, when combined with dynamic allocation, this prevents recovery from determining whether or not a set of bytes contains an LSN (since the usage varies over time). This leads to a *second* write-ahead log, incurring significant overhead [3, 21].

Segment-oriented recovery avoids this and supports write back caching (Section 7.2). In the case above, the page has different LSNs for A and B, but neither LSN is explicitly stored. Instead, recovery estimates the LSNs and recovers A and B independently; each object is its own segment.

3.3 Log Reordering

Having an LSN on each page also makes it difficult to reorder log entries, even between independent transactions. This interferes with mechanisms that prioritize important requests, and as with the buffer manager, tightly couples the log to the application, increasing synchronization and communication overheads.

In theory, all independent log entries could be reordered, as long as the order within objects and within transactions (e.g. the commit record) is maintained. However, in general even updates in two independent transactions cannot be reordered because they might share pages. Once an LSN is assigned to log entries on a shared page, the order of the independent updates is fixed.

With segment-oriented recovery we do not need to even *know* the LSN at the time of a page update, and can assign LSNs later if we choose. In some cases we assign LSNs at the time of writing the log to disk, which allows us to place high-priority entries at the *front* of the log buffer. Section 7.3 presents the positive impact this has on high-priority transactions. Before journaling was common, local filesystems supported such reordering. The Echo [23] distributed filesystem preserved these optimizations by layering a cache on top of a no-steal, non-transactional journaled filesystem.

Note that for dependent transactions, higher-level locks (isolation) constrain the order, and the update will block before it creates a log entry. Thus we are reordering transactions only in ways that preserve serializability.

3.4 Distributed recovery

Page-oriented recovery leads to a tight coupling between the application, the buffer manager and the log manager. Looking again at Figure 2, we note that the buffer manager must hold the latch across the call to the log manager so that it can atomically update the page with the correct LSN.

The tight coupling might be fine on a traditional single core machine, but it leads to performance issues when distributing the components to different machines and to a lesser extent, to different cores. Segment-oriented recovery enables simpler and looser coupling among components.

- Write back caching reduces communication between the buffer manager and application, since the communication occurs only on cache eviction.
- There is no need to latch the page during an update, since there is no shared state. (Races within one object are handled by higher-level locking.) Thus calls to the buffer manager and log manager can be asynchronous, hiding network latency.
- The use of natural layouts for large objects allows DMA and zero-copy I/O in the local case. In the distributed case, this allows application data to be written without copying the data and the LSNs to the same machine.

In turn, the ability to distribute these components means that they can be independently sized, partitioned and replicated. It is up to the system designer to choose partitioning and replication schemes, which components will coexist on the same machines, and to what extent calls to the underlying network primitives may be amortized and reordered.

This allows for very flexible large-scale write-ahead logging as a service for cloud computing, much the same way that two-phase commit or Paxos [18] are useful services.

3.5 Benefits from Pages

Pages provide benefits that complement segment-based approaches. They provide a natural unit for partitioning storage for use by different components; in particular, they enable the use of page headers that describe the layout of information on disk. Also, data structures such as B-trees are organized along page boundaries. This guarantees good locality for data that is likely to be accessed as a unit.

Furthermore, some database operations are significantly less expensive with page-oriented recovery. The most important is *page compaction*. Systems with atomic pages can make use of *physiological* updates that examine metadata, such as on-page tables of slot offsets. To compact such a page, page-based systems simply pin the page, defragment the page's free space, then unpin the page. In contrast, segment-based systems cannot rely on page metadata at redo and record such modifications in the log.

It may also make sense to build a B-tree using pages for internal nodes, and segments for the leaves. This would allow index nodes to benefit from physiological logging, but would provide high concurrency updates, reduced fragmentation and the other benefits of segments for the operations that read and write the data (as opposed to the keys) stored in the tree.

Page-oriented recovery simplifies the buffer manager because all pages are the same size, and objects do not span pages. Thus, the buffer manager may place a page at any point in its address space, then pass that pointer to the code interested in the page. In contrast, segment boundaries are less predictable and may change over time. This makes it difficult for the buffer manager to ensure that segments are contiguous in memory, although this problem is less serious with modern systems and large address spaces. Because pages and segments have different advantages, we are careful to allow them to safely coexist.

4. SEGMENT-BASED RECOVERY

This section provides an overview of ARIES and segments, and sketches a possible implementation of segment-based storage. This implementation is only one variant of our approach, and is designed to highlight the changes made by our proposal, not explain how to best use segments. Section 5 presents segments in terms of invariants that encompass a wide range of implementations.

Write-ahead logging systems consist of four components:

- The *log file* contains an in-order record of each operation. It consists of entries that contain an LSN (the offset into the log), the id of the transaction that generated the entry, which segment (or object) the entry changed, a boolean to show if the segment contains an LSN, and enough information to allow the modification to be repeated (we treat this as an operation implemented by the entry, e.g., `entry->redo()`). Recent entries may still reside in RAM, but older entries are stored on disk. Log *truncation* limits the log's size by erasing the earliest entries once they are no longer needed.
- The *application cache* is not part of the storage implementation. Instead, it is whatever in-memory representation the application uses to represent the data. It is often overlooked in descriptions of recovery algorithms; in fact, database implementations often avoid such caches entirely.
- The *buffer manager* keeps copies of disk pages in main memory. It provides an API that tracks LSNs and applies segment changes from the application cache to the buffers. In traditional ARIES, it presents a coherent view of the data. *Coherent*[1] means that changes are reflected in log order, which means that reads from the buffer manager immediately reflect updates performed by the application. Segment-based recovery allows applications to log updates (and perhaps update their own state), then defer and reorder the writes to the buffer manager. This leads to *incoherent* buffer managers that may return stale, contradictory data to the application. It is up to the application to decide when it is safe to read recently updated segments.
- The *page file* backs the buffer manager on disk and is incoherent. ARIES (and our example implementation) manipulates entire pages at a time; though segment-based systems could manipulate segments instead.

In page-based systems, each page is exactly one segment. Segment-based systems relax this and define segments to

[1] *Coherent* refers to a set of invariants analogous to those ensured by cache coherency protocols.

```
if(s->lsn_volatile <= log_stable) {
  write(s);
  s->lsn_stable = infinity;
  s->lsn_volatile = 0;
}
```

(a) Flush segment s to disk

```
s->lsn_stable =
   min(s->lsn_stable, entry->lsn);
s->lsn_volatile =
   max(s->lsn_volatile, entry->lsn);
entry->redo(s);
```

(b) Apply log entry to segment s

```
op_lsn = min<lsn_of_current_operations>;
t_lsn = min<start_lsn_of_current_xacts>;
s_lsn = min<segments->lsn_stable>;
log->truncate(min(op_lsn,t_lsn,s_lsn));
```

(c) Truncate log

Figure 3: Runtime operations for a segmented buffer manager. Page based buffer managers are identical, except their operations work against pages, causing **(b)** to split updates into multiple operations.

be arbitrary sets of *individually updatable* bytes; flushing a segment to disk cannot inadvertently change bytes outside the segment, even during a crash. There may be many higher-level objects per segment (records in a B-tree node) or many segments per object (arbitrary-length records). In both cases, storage deals with updates to one segment at a time.

Crucially, segments decouple application primitives (redo entries) from buffer management (disk operations). Regardless of whether the buffer manager provides a page or segment API, the data it contains is organized in terms of segments that represent higher level objects and are backed by disk sectors.

With a page API, updates to segments that span pages pin each page, manipulate a piece of the segment, then release the page. This works because blind writes will repair any torn (partially updated) segments, and because we assume that higher level code will latch segments as they are being written.

The key idea is to use segments to decouple updates from pages, allowing the application to choose the update granularity. This allows the requests to be reordered without regard to page boundaries.

The primary changes to forward operation relate to LSN tracking. Figure 3 describes a buffer manager that works with segments; paged buffer managers are identical, except that LSN tracking and other operations are per page, rather than per segment. `s->lsn_stable` is the first LSN that changed the in-memory copy of a page; `s->lsn_volatile` is the latest such value. If a page contains an LSN, then flushing it to disk sets the on-disk LSN to `s->lsn_volatile`.

If updates are applied in order, `s->lsn_stable` will only be changed when the page first becomes dirty. However, with reordering every update must check the LSN.

Write-ahead is enforced at page flush, which compares `s->lsn_volatile` to `log_stable`, the LSN of the most recent log entry to reach disk.

Truncation uses `s->lsn_stable` to avoid deleting log entries that recovery would need in order to bring the on-disk version of the page up-to-date. Because of reordering, truncation must also consider updates that have not reached the buffer manager. It also must avoid deleting undo entries that were produced by incomplete transactions.

4.1 Recovery

Like ARIES, segment-based recovery has three phases:

1. *Analysis* examines the log and constructs an estimate of the buffer manager's contents at crash. This allows later phases to ignore portions of the log.
2. *Redo* brings the system back into a state that existed before crash, including any incomplete transactions. This process is called *repeating history*.
3. *Undo* rolls back incomplete transactions and logs *compensation* records to avoid redundant work due to multiple crashes.

Also like ARIES, our approach supports *steal/no-force*. The actions performed by log entries are constrained to *physical redo*, which can be applied even if the system is inconsistent, and *logical undo*, which is necessary for concurrent transactions. Logical undo allows transactions to safely roll back after the underlying data has changed, such as when another transaction's B-tree insertion has rebalanced a node.

Hybrid redo

```
foreach(redo entry) {
  if(entry->clears_contents())
    segment->corrupt = false;
  if(entry->is_lsn_free()) {
    entry->redo(segment);
  } else if(segment->LSN < entry->LSN) {
    segment->LSN = entry->LSN
    error = entry->redo(segment);
    if(error) segment->corrupt = true;
  }
}
```

Unlike ARIES, which uses `segment->LSN` to ensure that each redo is applied *exactly once*, recovery always applies LSN-free redos, guaranteeing they reach the segment *at-least-once*. Hybrid systems, which allow ARIES and segments to coexist, introduce an additional change; they allow redo to temporarily corrupt pages.

This happens because segments store application data where ARIES would store an LSN and page header, leaving redo with no way to tell whether or not to apply ARIES-style entries. To solve this problem, hybrid systems zero out pages that switch between the two methods:

Switch page between ARIES and segment-based recovery

```
log(transaction id, segment id, new page type);
clear_contents(segment);
initialize_page_header(segment, new page type);
```

This ensures that recovery repairs any corruption caused by earlier redos.

4.2 Examples

We now present pseudocode for segment-based indexes and large objects.

Insert value into B-Tree node

```
make in-memory preimage of page
insert value into M'th of N slots
log (transaction id, page id, binary diff of page)
```

Segment-based indexes must perform blind writes during redo. Depending on the page format and fragmentation,

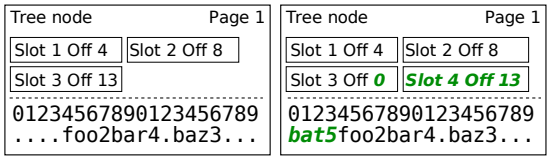

Figure 4: An internal tree node, before and after the pair (key="bat", page=5) is inserted.

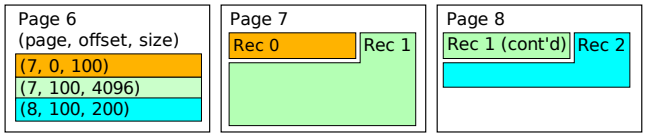

Figure 5: Records stored as segments. Colors correspond to (non-contiguous) bytes written by a single redo entry.

these entries could be relatively compact, as in Figure 4, or they could contain a preimage and postimage of the entire page, as would be the case if we inserted a longer key in Figure 4. In contrast, a conventional approach would simply log the slot number and the new value.

B-Tree concurrency is well-studied [20, 24], and largely unaffected by our approach. However, blind writes can incur significantly higher log overhead than physiological operations, especially for index operations. Fortunately, the two approaches coexist.

Update N segments

```
min_log = log->head

Spawn N parallel tasks; for each update:
  log (transaction id, offset, preimage, postimage)
Spawn N parallel tasks; for each update:
  pin and latch segment, s
  update s
  unlatch s
  s->lsn_stable = min(s->lsn_stable, min_log);

Wait for the 2N parallel tasks to complete
max_log = log->head

Spawn parallel tasks; for each segment, s:
  s->lsn_volatile = max(s->lsn_volatile, max_log);
  unpin s;
```

The latch is optional, and prevents concurrent access to the segment.[2] The pin prevents page flushes from violating the write-ahead invariant before `lsn_volatile` is updated. A system using the layout in Figure 5 and a page-based buffer manager would pin pages rather than segments and rely on higher level code to latch the segment.

Since the segments may happen to be stored on the same page, conventional approaches apply the writes in order, alternating between producing log entries and updating pages. Section 7 shows that this can incur significant overhead.

5. RECOVERY INVARIANTS

This section presents segment-based storage and ARIES in terms of first-order predicate logic. This allows us to prove the correctness of concurrent transactions and allocation. Unlike Kuo's proof [17] for ARIES, we do not present or prove correct a set of mechanisms that maintain our invariants, nor do we make use of I/O automata. Also unlike that work, we cover full, concurrent transactions and latching; two often misunderstood aspects of ARIES that are important to system designers.

5.1 Segments and objects

This paper uses the term *object* to refer to a piece of data that is written without regard to the contents of the rest of the database. Each object is physically backed by a set of *segments*: atomically logged, arbitrary length regions of disk. Segments are stored using *machine primitives*[3]; we assume the hardware is capable of updating segments independently, perhaps with the use of additional mechanisms. Like ARIES, segment-based storage is based on multi-level recovery [33], which imposes a nested structure upon objects; the nesting can be exploded to find all of the segments.

Let s denote an address, or set of addresses, i.e., a segment, and l denote the LSN of a log entry (an integer). Then, define s_l to be the value of that segment after applying a prefix of the log to the initial value of s:

$$s_l = log_l(log_{l-1}(...(log_1(s_0))))$$

Let s_t^{mem} be the value stored in the buffer manager at time t or $\perp$ if the segment is not in the buffer manager. Let s_t^{stable} be the value on disk. If $s_t^{mem} = s_t^{stable}$ or $s_t^{mem} = \perp$, then we say s is *clean*. Otherwise, s is *dirty*.

Finally, $s_t^{current}$ is the value stored in s:

$$s_t^{current} = \begin{cases} s_t^{mem} & \text{if } s_t^{mem} \neq \perp \\ s_t^{stable} & \text{otherwise} \end{cases}$$

Systems with coherent buffer managers maintain the invariant that $s_t^{current} = s_{l(t)}$, where $l(t)$ is the LSN of the most recent log entry at time t. Incoherent systems allow $s_t^{current}$ to be stale, and maintain the weaker invariant that $\exists\ l' \leq l(t) : s_t^{current} = s_{l'}$.

A *page* is a range of contiguous bytes with pre-determined boundaries. Although pages contain multiple application-level objects, if they are updated atomically then recovery treats them as a single segment/object. Otherwise, for the purposes of this section, we treat them as an array of single-byte segments. A *record* is an object that represents a simple piece of data, such as a tuple. Other examples of objects are indexes, schemas, or anything else stored by the system.

5.2 Coherency vs. Consistency

We define the set:

$$LSN(O) = \{l :\ O_l = O\} \tag{1}$$

to be the set of all LSNs l where O_l was equal to some version, O, of the object. With page-oriented storage, each page s contains an LSN, $s.lsn$. These systems ensure that $s.lsn \in LSN(s)$, usually by setting it to the LSN of the log entry that most recently modified the page. If s is not a page, or does not contain an explicit LSN, then $s.lsn = \perp$.

Object O is *corrupt* ($O = \top$) if it is a segment that never existed during forward operation, or if it contains a corrupt object:

$$\exists \text{ segment } s \in O : \forall \text{ LSN } l,\ s \neq s_l \tag{2}$$

[2] We assume `s->lsn_stable` and `s->lsn_volatile` are updated atomically.

[3] We take the term *machine* from the virtualization literature.

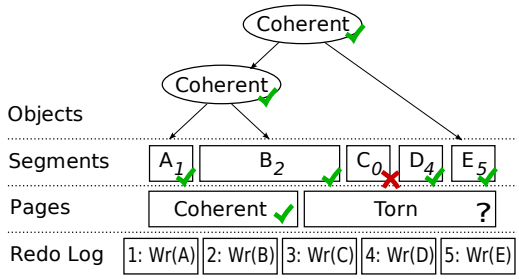

Figure 6: State of the system before redo; the data is incoherent (torn). Subscripts denote the most recent log entry to touch an object; Segment C is missing update 3. For the top level object $LSN(O) = \{5\}$. Segment B, the nested object and the coherent page have $LSN(O) = \{2, 3, 4, 5\}$. For the torn page, $LSN(O) = \emptyset$.

For the systems we consider, corruption only occurs due to faulty hardware or software, not system crashes. Repairing corrupted data is expensive, and requires access to a backed-up checkpoint of the database and all log entries generated since the checkpoint was taken. The process is analogous to recovery's redo phase; we omit the details.

Instead, the recovery algorithms we present here deal with two other classes of problems: torn (incoherent) data, and inconsistent data. An object O is *torn* if it is not corrupt and $LSN(O) = \emptyset$. In other words, the object was partially written to disk. Figure 6 shows some examples of torn objects as they might exist at the beginning of recovery.

An object O is *coherent* when it is in a state that arose during forward operation (perhaps mid-transaction):

$$\exists \text{ LSN } l : \forall \text{ object } o \in O, l \in LSN(o) \tag{3}$$

LEMMA 1. *O is coherent if and only if it is not torn.*

PROOF. *To show*

$$(\exists\, l : \forall\, s \in O, l \in LSN(s)) \iff (\exists\, l' \in LSN(O))$$

choose $l' = l$. For the $\Rightarrow$ case, each s is equal to s_l so O must be equal to O_l. By definition, $l \in LSN(O_l)$. The remaining case is analogous. □

Even though "torn" and "incoherent" are synonyms, we follow convention and reserve "torn" for discussions of partially written disk pages (or segments). We use "incoherent" when talking about multi-segment objects and the buffer manager.

An object is *consistent* if it is coherent at an LSN that was generated when there were no *in-progress* modifications to the object. Like objects, modifications are nested; a modification is in-progress if some of its sub-operations have not yet completed. As a special case; a transaction is an operation over the database; an ACID database is consistent when there are no in-progress transactions.

Physical operations can be applied when the database is incoherent, while logical operations rely on object consistency. For example, overwriting a byte at a known offset is a physical operation and always succeeds; traversing a multi-page index and inserting a key is a logical operation. If the index is inconsistent, it may contain partial updates normally protected by latches, and the traversal may fail.

Next, we explain how redo uses physical operations to bring the database to a coherent, but inconsistent state. This is not quite adequate for undo, which makes use of logical operations that can only be applied to consistent objects. Section 5.6 describes a runtime latching and logging protocol that guarantees undo's logical operations only encounter consistent objects.

5.3 The log and page files

Log entries are identified by an LSN, $e.lsn$, and specify an operation over a particular object, $e.object$, or segment, $e.segment$. If the entry modifies a segment, it applies a physical (or, in the case of ARIES, physiological) operation; if not, it applies a logical operation.

Log entries are associated with a transaction, $e.tid$, which is a set of operations that should be applied to the database in an atomic, durable fashion. The state of the log also includes three special LSNs: log_t^{trunc}, the beginning of the sequence that is stored on disk; log_t^{stable}, the last entry stored on disk; and $log_t^{volatile}$, the most recent entry in memory.

5.4 Write-ahead and checkpointing

Write-ahead ensures that updates reach the log file before they reach the page file:

$$\forall \text{ segment } s : \exists\, l \in LSN(s_t^{stable}) : l \leq log_t^{stable} \tag{4}$$

Log truncation and checkpointing ensure that all current information can be reconstructed from disk:

$$\forall \text{ segment } s, \exists\, l \in LSN(s_t^{stable}) : l \geq log_t^{trunc} \tag{5}$$

which ensures that the version of each object stored on disk existed at some point during the range of LSNs covered by the log.[4] Our proposed recovery scheme weakens this slightly; $\forall s$ that violate Equation 4 or 5:
$\exists$ redo $e : e.lsn \in \{l : log_t^{trunc} \leq l \leq log_t^{stable}\}$:

$$e.lsn \in LSN(e(\top)) \tag{6}$$

Where $e(\top)$ is the result of applying e to a corrupt segment. This will be needed for hybrid recovery (Section 6.2).

5.5 Three-pass recovery

Recall that recovery performs three passes; the first, analysis, is an optimization that determines portions of the log may be safely ignored.

The second pass, redo, is modified by segment based recovery. In both systems, the contents of the buffer manager are lost at crash, so at the beginning of redo, t_0:

$$\forall \text{ segment } s : s_{t_0}^{current} = s_{t_0}^{stable}$$

It then applies redo entries in log order, repeating history, and bringing the system into a coherent but perhaps inconsistent state. This maintains the following invariant:

$$\forall \text{ segment } s, \exists\, l \in LSN(s_t^{current}) : l \geq log\_cursor_t(s) \tag{7}$$

where $log\_cursor_t(s)$ is an LSN associated with the segment in question. During redo, $log\_cursor_t(s)$ monotonically increases from log_t^{trunc} to log_t^{stable}. Redo is parallelizable; each segment can be recovered independently. This allows online *media recovery*, which rebuilds corrupted pages by applying the redo log to a backed up copy of the database.

Redo assumes that the log is *complete*; $\forall$ segment s, lsn l,

$$s_{l-1} = s_l \vee (\exists\, e : e.lsn = l \wedge e.segment = s) \tag{8}$$

[4]For rollback to succeed, truncation must also avoid deleting entries from in-process transactions.

Either a segment is unchanged at a particular timestep, or there is a redo entry for that object at that timestep.

We now show that ARIES and segment-based recovery maintain the redo invariant (Equation 7). The hybrid approach is more complex and relies on allocation policies (Section 6.2).

5.5.1 ARIES redo strategy

ARIES applies a redo entry e with $l.lsn = log\_cursor(s)$ to a segment $s = e.segment$ if:

$$e.lsn > s.lsn$$

ARIES is able to apply this strategy because it stores an LSN from $LSN(s)$ with each segment (which is also a fixed-length page); therefore, $s.lsn$ is defined. Assuming the redo log is complete, this policy maintains the redo invariant.

This redo strategy maintains the further invariant that, before it applies e, $e.lsn-1 \in LSN(s)$; log entries are always applied to the same version of a segment.

5.5.2 Segment-based redo strategy

Our proposed algorithm always applies e. Since redo entries are *blind writes*, this yields an s such that $e.lsn \in LSN(s)$, regardless of the original value of the segment. Combined with completeness, this maintains the redo invariant.

5.5.3 Proof of redo's correctness

THEOREM 1. *At the end of redo, the database is coherent.*

PROOF. *From the definition of coherency (Equation 3), we need to show:*

$$\exists \text{ LSN } l : \forall \text{ object } O,\ l \in LSN(O)$$

By the definition of LSN(O) and an object, this is equivalent to:

$$\exists \text{ LSN } l : \forall \text{ segment } s \in O,\ l \in LSN(s)$$

Equations 4 and 7 ensure that:

$$\forall s, \exists\ l \in LSN(s) : log_t^{trunc} \leq log\_cursor_t(s) \leq l \leq log_t^{stable}$$

At the end of redo, $\forall s,\ log\_cursor_t(s) = l = log_t^{stable}$*, allowing us to reorder the universal and existential quantifiers.* □

The third phase of recovery, *undo* assumes that redo leaves the system in a coherent state. Since the database is coherent at the beginning of undo, we can treat transaction rollbacks during recovery in the same manner as rollbacks during forward operation. Next we prove rollback's correctness, concluding our treatment of recovery.

5.6 Transaction rollback

Multi-level recovery is compatible with concurrent transactions and allocation, even in the face of rollback. This section presents a special case of multi-level recovery: a simple, correct logging and latching scheme (Figure 7).

Like any other concurrent primitive, actions that manipulate transactional data temporarily break then restore various invariants as they execute. While such invariants are broken, other transactions must not observe the intermediate, inconsistent state.

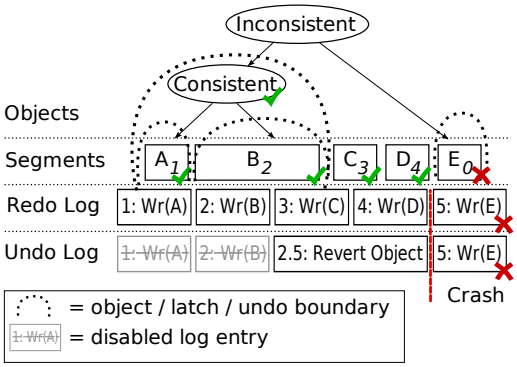

Figure 7: State of the system before undo; the data is coherent, but inconsistent. At runtime, updates hold each latch while manipulating the corresponding object, and release the latch when they log the undo. This ensures that undo entries never encounter inconsistent objects.

Recall that the definition of *coherent* (Equation 3) is based on nestings of recoverable objects. One approach to concurrent transactions obtains a latch on each object before modifying sub-objects, and then releases the latch before returning control to higher level operations. Establishing a partial ordering over the objects defines an ordering over the latches, guaranteeing that the system will not deadlock due to latch requests [13].

By construction, this scheme guarantees that all unlatched objects have no outstanding operations, and are therefore consistent. Atomically releasing latches and logging undo operations ties the undo to a point in time when the object was consistent; rollback ensures that undo operations will only be applied at such times. This latching scheme is more restrictive than necessary, but simplifies the implementation of logical operations [29]. More permissive approaches [20, 24] expose object state mid-operation.

The correctness of this scheme relies on the semantics of the undo operations. In particular, some are commutative (inserting x and y into a hashtable), while others are not ($z := 1$, $z := 2$). All operations from outstanding transactions must be commutative:

$\forall$ undo entry $e, f : e.tid \neq f.tid$,

$$o = e.object = f.object \Rightarrow e(f(o)) = f(e(o)) \qquad (9)$$

To support rollback, we log a logical undo for each higher level object update and a physical undo for each segment update. Each registration of a higher level undo *invalidates* lower level logical and physical undos, as does transaction commit. Invalidated undos are treated as though they no longer exist.[5] In addition to the truncation invariant for redo entries Equation 5, truncation waits for undo entries to be invalidated before deleting them. This is easily implemented by keeping track of the earliest LSN produced by ongoing transactions.

This, combined with our latching scheme guarantees that any violations of Equation 9 are due to two transactions directly invoking two non-commutative operations. This is a special case of *write-write conflicts* from the concurrency

[5]ARIES and segment-based recovery make use of logging mechanisms such as *nested top actions* and *compensation log records* to invalidate undo entries; we omit the details.

	Log preimage		Safety		Reuse before commit	
	Free	Alloc	LSN	Segment	Other xact	Same
1	Y		Y	Y	Y	Y
2		Y	Y	Y		Y
3		XOR	Y			Y
4	Never		Y	Y		

Figure 8: Allocation strategies.

control literature; in the absence of such conflicts, Equation 9 holds and the results of undo are unambiguous.

If we further assume that a concurrency control mechanism ensures the transactions are serializable, and if the undos are indeed the logical inverse of the corresponding forward operations, then rolling back a transaction places the system in a state logically equivalent to the one that would exist if the transaction were never initiated. This comes from the commutativity property in Equation 9.

Although concurrent data structure implementations are beyond the scope of this paper, there are two common approaches for dealing with lower-level conflicts. The first raises the level of abstraction before undoing an operation. For example, two transactions may update the same record while inserting different values into a B-tree. As each operation releases its latch, it logs an undo that will invoke the B-tree's "remove()" method instead of directly restoring the record. The second approach avoids lower-level conflicts. For example, some allocators guarantee space will not be reused until the transaction that freed the space commits.

6. ALLOCATION

The prior section treated allocation implicitly. A single object named the "database" spanned the entire page file, and allocation and deallocation were simply special operations over that object. In practice, recovery, allocation and concurrency control are tightly coupled. This section describes some possible approaches and identifies an efficient set that works with page- and segment-based recovery.

Transactional allocation algorithms must avoid unrecoverable states. In particular, reusing space or addresses that were freed by ongoing transactions leads to deadlock when those transactions rollback, as they attempt to reclaim the resources that they released. Unlike a deadlock in forward operation, deadlocks during rollback either halt the system or lead to cascading aborts.

Allocation consists of two sets of mechanisms. The first avoids unsafe conflicts by placing data appropriately and avoiding reuse of recently released resources. Data placement is a widely studied problem, though most discussions focus on performance. The second determines when data is written to log, ensuring that a copy of data freed by ongoing transactions exists somewhere in the system. Figure 8 summarizes four approaches.

The first two strategies log preimages, incurring the cost of extra logging; the fourth waits to reuse space until the transaction that freed the space commits. This makes it inappropriate for indexes and transactions that free space for immediate reuse.

The third option (labeled "XOR") refers to any *differential logging* [19] strategy that stores the new value as a function of the old value. Although differential updates and segment storage can coexist, differential page allocation is incompatible with our approach.

Differential logging was proposed as a way of increasing concurrency for main memory databases, and must apply log entries exactly once, but in any order. In contrast, our approach avoids the exactly once requirement, and is still able to parallelize redo (though to a lesser extent).

Logging preimages allows other transactions to overwrite the space that was taken up by the old object. This could happen due to page compaction, which consolidates free space on the page into a single region. Therefore, for pages that support reorganization, logging preimages at deallocation is the simplest approach.

For entire pages, or segments with unchanging boundaries, issues such as page compaction do not arise, so there is little reason to log at deallocation; instead a transaction can log preimages before reusing space it freed, or can avoid logging preimages altogether.

6.1 Existing hybrid allocation schemes

Recall that, without the benefit of per page version numbers, there is no way for redo to ensure that it is updating the correct version of a page. We could simply apply each redo entry in order, but there is no obvious way to decide whether or not a page contains an LSN. Inadvertently applying a redo to the wrong type of page corrupts the page.

Lotus Notes and Domino address the problem by recording synchronous page flushes and allocation events in the log, and adding extra passes to recovery [25]. The recovery passes ensure that page allocation information is coherent and matches the types of the pages that had made it to disk at crash. They extended this to multiple legacy allocation schemes and data types at the cost of great complexity [25].

Starburst records a table of current on-disk page maps in battery-backed RAM, skipping the extra recovery passes by keeping the appropriate state across crashes [4].

6.2 Correctness of hybrid redo

Here we prove Theorem 1 (redo's correctness) for hybrid ARIES and segment-based recovery. The hybrid allocator zeros out pages as they switch between LSN-free and segment-based formats. Also, page-oriented redo entries are only generated when the page contains an LSN, and segment-oriented redos are only generated when the page is LSN-free:

$$e.lsn\_free \iff lsn\_free(e.segment_{e.lsn}) \qquad (10)$$

THEOREM 2. *Hybrid redo leaves the database in a coherent state*

PROOF. *Equations 4 and 5 tell us each segment is coherent at the beginning of recovery. Although $lsn\_free(s)$ or $\neg lsn\_free(s)$ must be true, redo cannot distinguish between these two cases, and simply assumes the page starts in the format it was in when the beginning of the redo log was written.*

In the first case, this assumption is correct and redo will continue as normal for the pure LSN or LSN-free recovery algorithm. It will eventually complete or reach an entry that changes the page format, causing it to switch to the other redo algorithm. By the correctness of pure LSN and LSN-free redo (Section 5.5) this will maintain the invariant in Equation 7 until it completes.

In the second case, the assumption is incorrect. By Equation 10, the stable version of the page must have a different

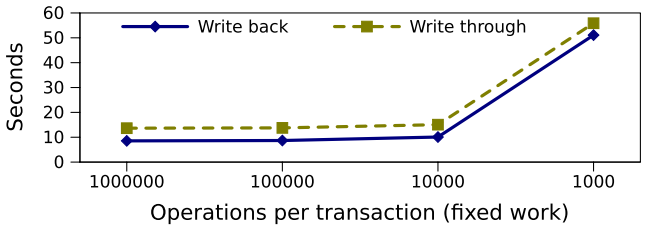

Figure 9: Time taken to transactionally update 10,000,000 `int` values. Write back reduces CPU overhead.

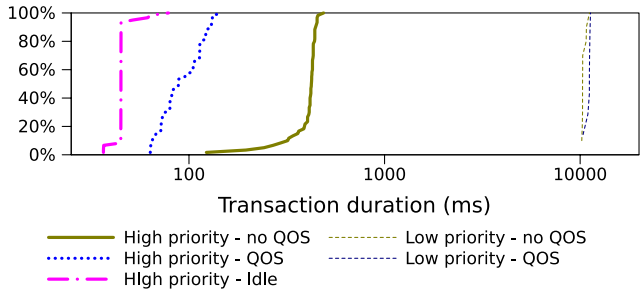

Figure 10: CDF of transaction completion times with and without log reordering.

type than it did when the redo entry was generated. Nevertheless, redo applies all log entries to the page, temporarily corrupting it. The write-ahead and truncation invariants, and log completeness (Equations 4, 5, and 8) guarantee that the log entry that changed the page's format is in the redo log. Once this entry, e, is encountered, it zeros out the page, repairing the corruption and ensuring that $e.lsn \in LSN(s)$, (Equation 6). At this point, the page format matches the current log entry, reducing this to the first case. □

7. DISCUSSION AND EVALUATION

Our experiments were run on an AMD Athlon 64 Processor 3000+ with a 1TB Samsung HD103UJ with write caching disabled, running Linux 2.6.27, and Stasis r1156.

7.1 Zero-copy I/O

Most large object schemes avoid writing data to log, and instead force-write data to pages at commit. Since the pages contain the only copy of the data in the system, applying blind writes to them would corrupt application data. Instead, we augment recovery's analysis pass, which already infers that certain pages are up-to-date. When a segment is allocated for force-writes, analysis adds it to a known-updated list, and removes it when the segment is freed.

This means that analysis' list of known-updated pages is now required for correctness, and must be guaranteed to fit in memory. Fortunately, redo can be performed on a per segment basis; if the list becomes too large, we partition the database, then perform an independent analysis and redo pass for each partition.

Zero-copy I/O complicates buffer management. If it is desirable to bypass the buffer manager's cache, then zero-copy writes must invalidate cached pages. If not, then the zero-copy primitives must be compatible with the buffer managers' memory layout. Once the necessary changes to recovery and buffer management are made, we expect the performance of large zero-copy writes to match that of existing file servers; increased file sizes decrease the relative cost of maintaining metadata.

7.2 Write caching

Read caching is a fairly common approach, both in local and distributed [10] architectures. However, distributed, durable write caching is more difficult and we are not aware of any commonly used systems.

Instead, each time an object is updated, it is marshaled then atomically (and synchronously) sent to the storage layer and copied to log and the buffer pool. This approach wastes both memory and time [29]. Even with minimal marshaling overheads, locating then pinning a page from the buffer manager decreases memory locality and incurs extra synchronization costs across CPUs.

To measure these costs, we extended Stasis with support for segments within pages, and removed LSNs from the header of such pages. We then built a simple application cache. To perform an LSN-free write, we append redo/undo entries to log, then update the application cache, causing the buffer manager to become incoherent. Before shutdown, we write back the contents of cache to the buffer manager.

To perform conventional write through, we do not set up the cache and instead call Stasis' existing record set method.

Because the buffer manager is incoherent, our optimization provides *no-force* between the application cache and buffer manager. In contrast, applications built on ARIES force data to the buffer pool at each update instead of once at shutdown. This increases CPU costs substantially.

The effects of extra buffer management overhead are noticeable even in the single-threaded case; Figure 9 compares the cost of durably updating 10,000,000 integers using transactions of varying size.

For small transactions, (less than about 10,000 updates) the cost of force writing the log at commit dominates performance. For larger transactions, most of the time is spent on asynchronous log writes and on buffer manager operations. We expect the gap between write back and write through to be higher in systems that marshal objects (instead of raw integers), and in systems with greater log bandwidth.

7.3 Quality of service

We again extend Stasis, this time allowing each transaction to optionally register a low-priority queue for its segment updates. To perform a write, transactions pin and update the page, then submit the log entry to the queue. As the queue writes back log entries, it unpins pages. We use these primitives to implement a simple quality of service mechanism. The disk supports a fixed number of synchronous writes per second, and Stasis maintains a log buffer in memory. Low priority transactions ensure that a fraction of Stasis' write queue is unused, reserving space for high-priority transactions. A subtle, but important detail of this scheme is that, because transactions unlatch pages before appending data to log, backpressure from the logger decreases page latch contention; page-based systems hold latches across log operations, leading to increased contention and lower read throughput.

For our experiment, we run "low priority" bulk transactions that continuously update records with no delay, and "high priority" transactions that only update a single record,

Storage algorithm	Small workload		Large workload	
	Local	Network	Local	Network
Pages	0.866s	61s	10.86s	6254s
Segments	0.820s	26s	5.893s	105s
Segs. (bulk messages)	”	8s	”	13s

Figure 11: Comparison of segment and page based recovery with simulated network latency. The small workload runs ten transactions of 1000 updates each; the large workload runs ten of 100,000 each.

but run once a second. This simulates a high-throughput bulk load running in parallel with low-latency application requests.

Figure 10 plots the cumulative distribution function of the transactions' response times. With log reordering (QOS) in place, worst case response time for high priority transactions is approximately 140ms; "idle" reports high priority transaction performance without background tasks.

7.4 Recovery for distributed systems

Data center and cloud computing architectures are often provisioned in terms of *applications*, *cache*, *storage* and *reliable queues*. Though their implementations are complex, highly available approaches with linear scalability are available for each service.

However, scaling these primitives is expensive, and operations against these systems are often heavy-weight, leading to poor response times and poor utilization of hardware. Write reordering and write caching help address these bottlenecks. For our evaluation, we focused on reordering requests to write to the log and writing-back updates to the buffer manager.

We modified Stasis with the intention of simulating a network environment. We add $2ms$ delays to each request to append data to Stasis' log buffer, or to read or write records in the buffer manager. We did not simulate the overhead of communicating LSN estimates between the log and page storage nodes. We ran our experiment with both write back and write reordering enabled (Figure 11), running one transaction at a time. For the "bulk messages" experiments, we batch requests rather than send one per network round trip.

For small transactions, the networked version is roughly ten times slower than the local versions, but approximately 20 times faster than a distributed, page-oriented approach. As transaction sizes increase, segment-based recovery is better able to amortize network round trips due to log and buffer manager requests, and network throughput improves to more than 400 times that of the page-based approach. As above, the local versions of these benchmarks are competitive with local page-oriented approaches, especially for long transactions.

A true distributed implementation would introduce additional overheads and opportunities for improved scalability. In particular, replication will allow the components to cope with partial failure and partitioning should provide linear scalability within each component. How such an approach interacts with real-world workloads is an open question.

As with any other distributed system, there will be tradeoffs between consistency and performance; we suspect that durability based upon distributed write-ahead logging will provide significantly greater performance and flexibility than systems based on synchronous updates of replicas.

8. RELATED WORK

Here, we focus on other approaches to the problems we address. First we discuss systems with support for log reordering, then we discuss distributed write-ahead logging.

Write reordering mechanisms provide the most benefit in systems with long running, non-durably committed requests. Therefore, most related work in this area comes from the filesystem community. Among filesystems, our design is perhaps most similar to Echo [23]. Its *write-behind* queues provide rich write reordering semantics and are a non-durable version of our reorderable write-ahead logs.

FeatherStitch [12] introduces filesystem *patches*; sets of atomic block writes (blind writes) with ordering constraints, and allows the block scheduler and applications to reorder patches. Rather than provide concurrent transactions, it provides filesystem semantics and a `pg_sync` mechanism that explicitly force-writes a patch and its dependencies to disk.

Although our distributed performance results are promising, designing a complete, scalable and fault tolerant storage system from our algorithm is non-trivial. Fortunately, the implementation of each component in our design is well understood. Read only caching technologies such as memcached [10] would provide a good starting point for linearly scalable write back application caches. Main-memory database techniques are increasingly sophisticated, and support compression, superscalar optimizations, and isolation.

Scalable data storage is also widely studied. Cluster hash tables [11], which partition data across independent index nodes, and Boxwood [22], which distributes indexes across clusters, are two extreme points in the scope of possible designs. A third approach, Sinfonia [1], has nodes expose a linear address space, then performs *minitransactions*; essentially atomic bundles of test and set operations against these nodes. In contrast, page writeback allows us to apply many transactions to the storage nodes with a single network round trip, but relies on a logging service.

A number of reliable log services are already available, including ones that scale up to data center and Internet scale workloads. In the context of cloud computing, indexes such as B-Trees have been implemented on top of Amazon SQS (a scalable, reliable log) and S3 (a scalable record store) using purely logical redo and undo; those approaches require write-ahead logging or other recovery mechanisms at each storage node [3]. Application specific systems also exist, and handle atomicity in the face of unreliable clients [27].

A second cloud computing approach is extremely similar to our distributed proposal, but handles concurrency and reordering with explicit per object LSNs and exactly-once redo [21]. Replicas store objects in durable key-value stores that are backed by a second, local, recovery mechanism. An additional set of mechanisms ensures that recovery's redo phase is idempotent. In contrast, implementing idempotent redo is straightforward in segment-based systems.

9. CONCLUSION

Segment-based recovery operates at the granularity of application requests, removing LSNs from pages. It brings request reordering and reduced communication costs to concurrent, steal/no-force database recovery algorithms. We presented ARIES-style and segment-based recovery in terms of the invariants they maintain, leading to a simple proof of their correctness.

The results of our experiments suggest segment-based recovery significantly improves performance, particularly for transactions run alongside application caches, run with different priorities, or run across large-scale distributed systems. We have not yet built practical segment-based storage. However, we are currently building a number of systems based on the ideas presented here.

10. ACKNOWLEDGMENTS

Special thanks to our shepherd, Alan Fekete for his help correcting and greatly improving the presentation of this work.

We would also like to thank Peter Alvaro, Brian Frank Cooper, Tyson Condie, Joe Hellerstein, Akshay Krishnamurthy, Blaine Nelson, Rick Spillane and the anonymous reviewers for suggestions regarding earlier drafts this paper.

Our discussions with Phil Bohannon, Catharine van Ingen, Jim Gray, C. Mohan, P.P.S. Narayan, Mehul Shah and David Wu clarified these ideas, and brought existing approaches and open challenges to our attention.

11. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[3] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.

[4] L. Cabrera, J. McPherson, P. Schwarz, and J. Wyllie. Implementing atomicity in two systems: Techniques, tradeoffs, and experience. *TOSE*, 19(10), 1993.

[5] D. Chamberlin et al. A history and evaluation of system R. *CACM*, 24(10), 1981.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] R. A. Crus. Data recovery in IBM Database 2. *IBM Systems Journal*, 23(2), 1984.

[8] P. A. Dearnley. An investigation into database resilience. *Oxford Computer Journal*, July 1975.

[9] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *SOSP*, 1993.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, August 2004.

[11] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, 1997.

[12] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and Lei. Generalized file system dependencies. In *SOSP*, 2007.

[13] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. *Modelling in Data Base Management Systems*, pages 365–r394. North-Holland, Amsterdam, 1976.

[14] T. Greanier. Serialization API. In *JavaWorld*, 2000.

[15] T. Haerder and A. Reuter. Principles of transaction oriented database recovery—a taxonomy. *ACM Computing Surveys*, 1983.

[16] Hibernate. http://www.hibernate.org/.

[17] D. Kuo. Model and verification of a data manager based on ARIES. *TODS*, 21(4), 1996.

[18] L. Lamport. Paxos made simple. *SIGACT News*, 2001.

[19] J. Lee, K. Kim, and S. Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory databases. In *ICDE*, 2001.

[20] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *TODS*, 1981.

[21] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[22] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[23] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *TOCS*, May 1994.

[24] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control multiaction transactions operating on B-tree indexes. In *VLDB*, 1990.

[25] C. Mohan. A database perspective on Lotus Domino/Notes. In *SIGMOD Tutorial*, 1999.

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

[27] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Making a cloud provenance-aware. In *TAPP*, 2009.

[28] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In *OSDI*, 2006.

[29] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[30] M. Seltzer and M. Olsen. LIBTP: Portable, modular transactions for UNIX. In *Usenix*, January 1992.

[31] *SQL Server 2008 Documetation*, chapter Buffer Management. Microsoft, 2009.

[32] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for Unix. Technical Report SMLI TR-95-39, Sun Microsystems, 1995.

[33] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *PODS*, 1990.

[34] M. Widenius and D. Axmark. *MySQL Manual*.