Stasis: Flexible Transactional Storage

Russell C Sears

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-2
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-2.html

January 8, 2010

Stasis: Flexible Transactional Storage

by

Russell C. Sears

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2009

Stasis: Flexible Transactional Storage

Abstract

Stasis: Flexible Transactional Storage

by

Russell C. Sears

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

An increasing range of applications requires robust support for atomic, durable and concurrent transactions. Databases provide the default solution, but force applications to interact via SQL and to forfeit control over data layout and access mechanisms. In principle, a specialized database stack could be built for each application, but such approaches have proven to be impractical. We argue there is a gap between DBMSs and file systems that limits designers of data-oriented applications.

Stasis is a storage framework that incorporates ideas from traditional write-ahead logging algorithms and file systems. It provides applications with flexible control over data structures, data layout, robustness and performance. Stasis enables the development of unforeseen variants on transactional storage by generalizing write-ahead logging algorithms. Instead of implementing support for each new storage system from scratch, I have extended Stasis to provide specialized storage mechanisms to a wide variety of applications. It now provides cleaner semantics than similar application-specific approaches would, with significantly less source code than would be required by multiple separate storage implementations. In addition to the conventional write-ahead logging algorithms that Stasis was designed for, it now provides support for large objects, and for log-structured indexes. A number of other extensions, such as distributed recovery algorithms and snapshot-based recovery are under development.

This dissertation describes the range of data models and program architectures that have been commonly used in the past, and argues that Stasis is sufficiently general to support most storage applications. It then turns to a description of Stasis' high-level application interfaces and APIs that are designed to allow applications to add their own transactional data structures to Stasis. The performance of a number of such extensions is evaluated, showing that Stasis performs favorably relative to existing systems.

The dissertation then turns to a careful definition of Stasis' recovery algorithms, and provides a novel generalization of ARIES, the de facto *standard approach to transactional storage. The generalization is particularly promising in the context of distributed systems. Finally, it presents Stasis' lower-level interfaces, providing systems developers and application designers with the ability to tailor high-level transactional primitives to new types of storage hardware and operating system primitives. To the greatest extent possible, the ideas presented within are composable, allowing Stasis' simple implementation to support an unusually wide range of storage architectures.*

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank Eric Brewer for his support and input over the years, and my undergraduate research adviser, Bo Yuan. The lessons I learned from Bo and the rest of his lab at Ohio State provided me with invaluable experience and much of the motivation behind this work.

It has been a pleasure to work with Joe Hellerstein, and his students Peter Alvaro, Tyson Condie and Neil Conway. I have applied lessons learned from our discussions of Datalog and query processing to Stasis' implementation and portions of this dissertation; hopefully this effort has helped clarify the ideas contained within.

Anthony Joseph and Mike Franklin have provided invaluable feedback, as have my outside committee members, Ray Larson and Marti Hearst.

Stasis borrows ideas from many places, including fellow students and experts from the database and storage world. Mike Demmer pointed out that naive transactional storage systems needlessly double buffer objects and perform unnecessary marshaling operations on each update. He and Bowei Du implemented Oasys, and helped with my first attempt to apply segment-style optimizations to object stores (C. Mohan politely pointed out flaws in the approach a few months later, and has periodically offered me guidance ever since.)

During my time in graduate school, I was fortunate enough to be mentored by Catharine van Ingen, Jim Gray, Petros Maniatis, Mark Callaghan, and Nitin Parab. Jim suggested we use a resource manager to track dependencies within Stasis and provided feedback on the LSN-free recovery algorithms. I appreciate his mentorship and Catharine's. My understanding of storage performance issues, especially those related to changing hardware trends and long-term deployments are largely due to my time spent working with them. I implemented Rose under the direction of Mark; he provided the initial design and pointed me at the relevant literature.

I was also fortunate enough to work with Rick Spillane and the Stony Brook FSL on Story Book and a number of other Stasis-related endeavors.

Bill Weihl and Alan Fekete shepherded papers that contributed to this dissertation; their input certainly improved the presentation of this material.

Stasis was originally called LLADD, and the original code base was written by Jim Blomo, Jason Bayer and Jimmy Kittiyachavalit.

Mark Gritter, Gilad Arnold, Amir Kamil, Colleen Lewis, and Armin Primadi helped test early versions of Stasis. Also, I would like to thank Mert Akdere, Utkarsh Srivastava and Brian Cooper for their work on a successor to Rose.

Portions of this work were funded by a research grant from Sun Microsystems and an ARCS fellowship. Portions were performed at Intel Research Berkeley, Google, and Tintri.

Finally, I'd like to thank my wife, Dara for all of her support over the years, and quite a few last-minute proofreading sessions.

Chapter 1

Introduction

As our reliance on computing infrastructure increases, a wider range of applications requires robust data management. Traditionally, data management has been the province of database management systems (DBMSs), which are well suited to enterprise applications, but lead to poor support for systems such as web services, search engines, source code control systems, work-flow applications, bioinformatics, and scientific computing. These applications have complex transactional storage requirements, but do not fit well onto SQL or the monolithic approach of current databases. In fact, when performance matters these applications often avoid DBMSs and instead implement ad-hoc data management solutions (35; 41).

An example of this mismatch occurs with DBMS support for persistent objects. In a typical usage, an array of objects is made persistent by mapping each object to a row in a table (or sometimes multiple tables) (61) and then issuing queries to keep the objects and rows consistent. Also, for performance reasons, most systems must buffer two copies of the application's working set in memory. This is an awkward and inefficient use of RAM, and hence we claim that DBMSs do not support this task well.

Search engines and data warehouses in theory can use the relational model, but in practice need a very different implementation. Object-oriented, XML and streaming databases all have distinct conceptual models and underlying implementations.

Scientific computing, bioinformatics and document management systems tend to preserve old versions and track provenance. Thus they each have a distinct conceptual model. Bioinformatics systems perform computations over large, semi-structured databases. Relational databases support none of these requirements well. Instead, office suites, ad-hoc text-based formats and Perl scripts are used for data management (125) with mixed success (149).

Our hypothesis is that 1) each of these areas has a distinct top-down conceptual model (which may not map well to the relational model); and 2) there exists a bottom-up layered framework that can better support all of these models and others.

To explore this hypothesis, we present Stasis, a library that provides transactional storage at a level of abstraction as close to the hardware as possible. It can support special-purpose transactional storage models in addition to ACID database-style interfaces to abstract data models. Stasis incorporates techniques from both databases (e.g. write-ahead logging) and operating systems (e.g. zero-copy techniques).

Our goal is to combine the flexibility and layering of low-level abstractions typical for systems work with the complete semantics that exemplify the database field. By *flexible* we mean that Stasis can support a wide range of transactional data structures *efficiently*, and that it can support a variety of policies for locking, commit, buffer management and distributed operation. Also, it is extensible for new core operations and data structures. This flexibility allows it to support a wide range of systems and models.

By *complete* we mean full redo/undo logging that supports both no-Force, which provides durability with only log writes, and Steal, which allows dirty pages to be written out prematurely to reduce memory pressure. By complete, we also mean support for media recovery, which is the ability to roll forward from an archived copy, and support for error-handling, clusters, and multithreading. These requirements are difficult to meet and form the *raison d'être* for Stasis: the framework delivers these properties as reusable building blocks for systems that implement complete transactions.

1.1 Stasis is not a database

Database research has a long history, including the development of many of the technologies we exploit. This section explains why databases are fundamentally inappropriate tools for system developers, and covers some of the previous responses of the systems community. These problems have been the focus of database and systems researchers since databases were invented 45 years ago.

1.1.1 The database view

Recently, the database community has approached the limited range of DBMSs by either creating new top-down models, such as object-oriented, XML or streaming databases (28; 76; 88), or by extending the relational model (33) along some axis, such as new data types (129) or physical storage primitives. Earlier database systems such as CODASYL provided more general storage primitives than today's relational databases, but were notoriously complicated, both for end-users, and for the database implementers (87). We cover these approaches in more detail in Section 2.4 and in Part I.

An early survey of database implementations sought to enumerate the components used by database system implementers (11; 13). This survey was performed due to difficulties in extending database systems into new application domains. It divided internal database routines into two broad modules: *conceptual mappings* and *physical database models*. It is the responsibility of a database implementer to choose a set of conceptual mappings that implement the desired higher-level abstraction (such as the relational model). The physical data model is chosen to support efficiently the set of mappings that are built on top of it.

A conceptual mapping based on the relational model might translate a relation into a set of keyed tuples. If the database were going to be used for short, write-intensive and high-concurrency transactions (e.g. banking), the physical model would probably translate sets of tuples into an on-disk B-tree. In contrast, if the database needed to support long-running, read-only aggregation queries over high-dimensional data (e.g. data warehousing), a physical model that stores the data in a sparse array format would be more appropriate (30; 150). Although both kinds of databases are based upon the relational model they make use of different physical models in order to serve different classes of applications efficiently.

A basic claim of this work is that no known physical data model can efficiently support the wide range of conceptual mappings that are in use today. In addition to sets, objects, and XML, such a model would need to cover search engines, version-control systems, work-flow applications, and scientific computing, as examples. Similarly, a recent database paper argues that the "one size fits all" approach of DBMSs no longer works (130).

Instead of attempting to create such a unified model after decades of database research has failed to produce one, we opt to provide a bottom-up transactional toolbox that supports many models efficiently. This makes it easy for system designers to implement most data models that the underlying hardware can support, or to abandon the database approach entirely, and forgo a top-down model.

1.1.2 The systems view

The systems community has also worked on this mismatch, which has led to many interesting projects. Examples include alternative durability models such as QuickSilver (115), RVM (114), persistent objects (80), and persistent data structures (55; 84). We expect that Stasis would simplify the implementation of most if not all of these systems. Section 2.4 covers these in more detail.

Stasis is based upon the theory that a single, general-purpose, low-level storage system can support the full range of storage applications. In some sense, our hypothesis is trivially true in that there exists a bottom-up framework called the "operating system" that can implement all of the models. A famous database paper argues that it does so poorly (126). Our task is really to simplify the implementation of transactional systems through more powerful primitives that enable concurrent transactions with a variety of performance/robustness tradeoffs.

The closest system to ours in spirit is Berkeley DB, a highly successful lightweight alternative to conventional databases (119). At its core, it provides the physical database model (relational storage system (8)) of a conventional database server. In particular, it provides transactional (ACID) operations on B-trees, hash tables, and other access methods. It provides flags that let its users tweak aspects of the performance of these primitives, and selectively disable the features it provides.

Stasis has been used to build a number of transactional storage systems that Berkeley DB cannot efficiently support, such as systems that rely upon custom isolation semantics and log structured indexes. Berkeley DB is applicable to a narrower range of storage systems than Stasis because it makes a number of assumptions regarding workloads and low-level data representations. Thus, although Berkeley DB could be built on top of Stasis, Berkeley DB's data model and write-ahead logging system are too specialized to support Stasis.

1.1.3 The Stasis view

A careful examination of data-intensive applications, databases, persistent data structure implementations (such as (60; 139)) and other storage systems reveals that they are implemented using layers of abstraction, or *mappings*, with physical storage primitives at the bottom and conceptual

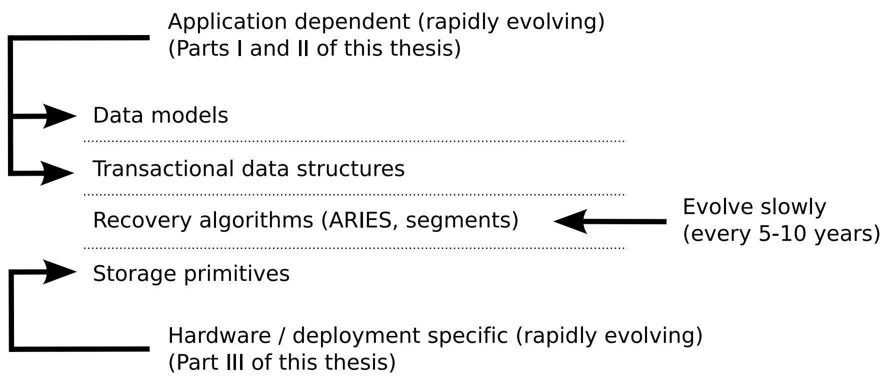

Application dependent (rapidly evolving)
(Parts I and II of this thesis)
Data models
Transactional data structures
Recovery algorithms (ARIES, segments)
Evolve slowly
(every 5-10 years)
Storage primitives
Hardware / deployment specific (rapidly evolving)
(Part III of this thesis)

Figure 1.1. Stasis' four levels of abstraction. Three of these evolve rapidly as new applications and hardware platforms are introduced. The fourth, recovery, changes less quickly.

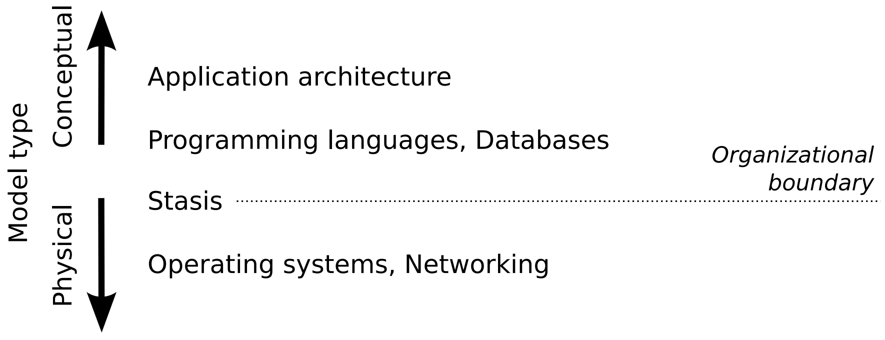


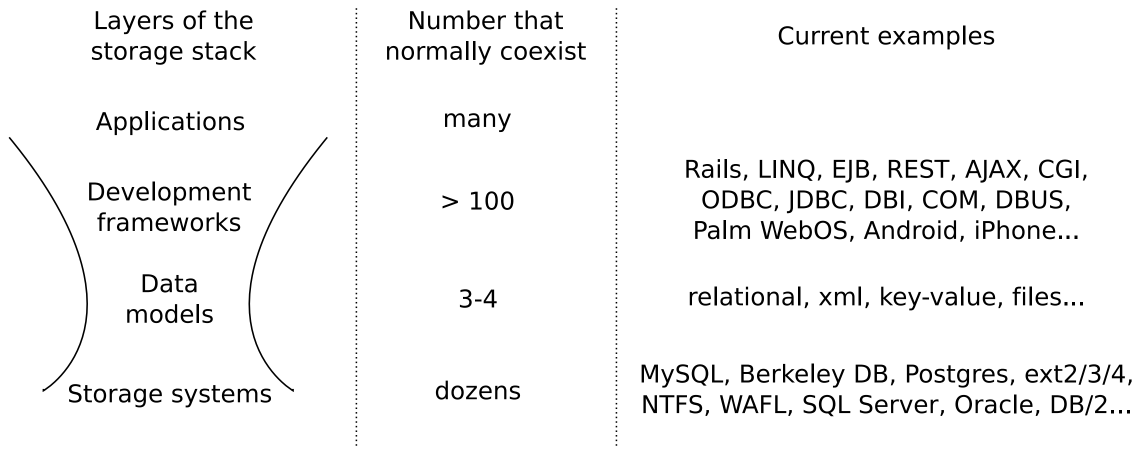
Figure 1.2. Stasis provides a clean separation of concerns for users with different areas of expertise.

models at the top. The extensible databases mentioned above collapsed these mappings into two layers: the conceptual model and the physical database representation.

Stasis is constructed from four layers; three layers are only loosely coupled, and must evolve rapidly to meet changing application demands and hardware changes. The fourth is a general-purpose recovery implementation designed to address as wide a range of applications as possible, and to be essentially unchanging. The insight behind Stasis' design is that much of the complexity and subtlety of transactional storage is due to recovery algorithms and a few reusable primitives in the other layers (such as allocation). The mechanisms that applications are likely to change are comparatively simple (Figure 1.1).

Encouragingly, Stasis' users tend to focus on a single module at a time; in fact users' backgrounds are a strong predictor of the layers they are interested in manipulating; operating system developers typically focus upon the lower levels, while database and programming language experts typically focus on data models and transactional data structure implementations (Figure 1.2). An organization that intends to extend multiple layers of Stasis discovered this layering independently, and divided its development process and organization chart along these lines. This suggests that Stasis successfully encapsulates higher levels of abstraction from lower-level details.

Figure 1.3 presents a sketch of the storage industry. Traditionally, a small number of data models served a wide range of program architectures and applications, and a handful of independent storage implementations provided each of the data models. As the range of applications has increased, data models and storage implementations have increasingly become bottlenecks against



Layers of the storage stack	Number that normally coexist	Current examples
Applications	many	
Development frameworks	> 100	Rails, LINQ, EJB, REST, AJAX, CGI, ODBC, JDBC, DBI, COM, DBUS, Palm WebOS, Android, iPhone...
Data models	3-4	relational, xml, key-value, files...
Storage systems	dozens	MySQL, Berkeley DB, Postgres, ext2/3/4, NTFS, WAFL, SQL Server, Oracle, DB/2...

Figure 1.3. A storage bottleneck. Each storage system is coupled coupled to the data model it provides, making it economically difficult for the industry to support many data models simultaneously. Because data models are scarce, it is difficult to introduce new types of applications, and the development frameworks that bridge application logic and the storage system are unduly complex.

further development (Chapter 6). Stasis' goal is to decouple storage mechanisms from data models. The hope is that, by allowing a wider range of developers to extend the lower levels of the storage stack, Stasis will be able to support a wider range of applications than current storage approaches, and will address a number of problems with current data management architectures.

1.2 Transactional pages

This section describes how Stasis implements transactions that are similar to those provided by relational database systems, which are based on transactional pages. The algorithms described in this section are not novel, and are in fact based on ARIES (95). However, they form the starting point for extensions and novel variants, which we cover in the next two sections.

As with other systems, Stasis' transactions have a multi-level structure. Multi-layered transactions were originally proposed as a concurrency control strategy for database servers that support high-level, application-specific extensions (142). In Stasis, the lower level of an operation provides atomic updates to regions of the disk. These updates do not have to deal with concurrency, but must update the page file atomically, even if the system crashes.

Higher-level operations span multiple pages by atomically applying sets of operations to the page file, recording their actions in the log and coping with concurrency issues. The loose coupling of these layers lets Stasis' users compose and reuse existing operations.

1.2.1 Non-concurrent transactions

This section provides the "Atomicity" and "Durability" properties for a single ACID transaction.[1] First we describe single-page transactions, then multi-page transactions. "Consistency" and "Isolation" are covered with concurrent transactions in the next section.

The insight behind transactional pages was that atomic page writes form a good foundation for full transactions. However, since page writes are no longer atomic, it might be better to think of these as transactional sectors.

The trivial way to achieve single-page transactions is to apply all of the updates to the page and then write it out on commit. The page must be pinned until commit to prevent write-back of uncommitted data, but no logging is required.

This approach performs poorly because we *force* the page to disk on commit, which leads to a large number of synchronous non-sequential writes. By writing redo information to the log before committing (write-ahead logging), we get no-Force transactions and better performance, since the synchronous writes to the log are sequential. Later, the pages are written out asynchronously, often as part of a larger sequential write.

After a crash, we have to apply the redo entries to those pages that were not updated on disk. To decide which updates to reapply, we use a per-page version number called the *log-sequence number* or *LSN*. Each update to a page increments the LSN, writes it on the page, and includes it in the log entry. On recovery, we load the page, use the LSN to figure out which updates are missing (those with higher LSNs), and reapply them.

Updates from aborted transactions should not be applied, so we also need to log commit records; a transaction commits when its commit record correctly reaches the disk. Recovery starts with an analysis phase that determines all of the outstanding transactions and their fate. The redo phase then applies the missing updates for committed transactions.

Pinning pages until commit also hurts performance, and could even affect correctness if a single transaction needs to update more pages than can fit in memory. A related problem is that with concurrency a single page may be pinned forever as long as it has at least one active transaction in progress all the time. Systems that support Steal avoid these problems by allowing pages to be written back early. This implies we may need to undo updates on the page if the transaction aborts, and thus before we can write out the page we must write the undo information to the log.

On recovery, the redo phase applies all updates (even those from aborted transactions). Then, an undo phase corrects stolen pages for aborted transactions. Each operation that undo performs is recorded in the log, and the per-page LSN is updated accordingly. In order to ensure progress even with crashes during recovery, special log records mark which actions have been undone, so they may be skipped during recovery in the future. We also use these records, called *Compensation Log Records (CLRs)* to avoid undoing actions that we intend to keep even when transactions abort.

Stasis allows user-defined operations that consist of an undo and a redo function. Each time an operation is invoked, a corresponding log entry is generated.

Given Steal/no-Force single-page transactions, it is relatively easy to build full transactions. To recover a multi-page transaction, we simply recover each of the pages individually. This works

[1]The "A" in ACID really means "atomic persistence of data," rather than "atomic in-memory updates," as the term is normally used in systems work; the latter is covered by "C" and "I" (53).

because Steal/no-Force completely decouples the pages: any page can be written back early (Steal) or late (no-Force).

1.2.2 Concurrent transactions

Two factors make it more complicated to write operations that may be used in concurrent transactions. The first is familiar to anyone that has written multi-threaded code: Accesses to shared data structures must be protected by latches (mutexes). The second problem stems from the fact that abort cannot simply roll back physical updates. Fortunately, it is straightforward to reduce this second, transaction-specific problem to the familiar problem of writing multi-threaded software.

To understand the problems that arise with concurrent transactions, consider what would happen if one transaction, A, rearranges the layout of a data structure. Next, another transaction, B, modifies that structure and then A aborts. When A rolls back, its undo entries will undo the changes that it made to the data structure, without regard to B's modifications. This is likely to cause corruption.

Two common solutions to this problem are *total isolation* and *nested top actions*. Total isolation prevents any transaction from accessing a data structure that has been modified by another in-progress transaction. An application can achieve this using its own concurrency control mechanisms, or by holding a lock on each data structure until the end of the transaction (by performing *strict two-phase locking* on the entire data structure). Releasing the lock after the modification, but before the end of the transaction, increases concurrency. However, it means that follow-on transactions that use the data may need to abort if this transaction aborts (*cascading aborts*).

Nested top actions avoid this problem. The key idea is to distinguish between the logical operations of a data structure, such as adding an item to a set, and internal physical operations such as splitting tree nodes. The internal operations do not need to be undone if the containing transaction aborts; instead of removing the data item from the page, and merging any nodes that the insertion split, we simply remove the item from the set as application code would—we call the data structure's *remove* method. That way, we can undo the insertion even if the nodes that were split no longer exist, or if the data item has been relocated to a different page. This lets other transactions manipulate the data structure before the first transaction commits.

In Stasis, each nested top action performs a single logical operation by applying a number of physical operations to the page file. Physical redo and undo log entries are stored in the log so that recovery can repair any temporary inconsistency that the nested top action introduces. Once the nested top action has completed, a logical undo entry is recorded, and a CLR is used to tell recovery and abort to skip the physical undo entries.

This leads to a mechanical approach for creating reentrant, concurrent operations:

1. Wrap a mutex around each operation. With care, it is possible to use finer-grained latches in a Stasis operation (96), but it is rarely necessary.
2. Define a *logical* undo for each operation (rather than a set of page-level undos). For example, this is easy for a hash table: the undo for *insert* is *remove*. The logical undo function should arrange to acquire the mutex when invoked by abort or recovery.
3. Add a "begin nested top action" right after mutex acquisition, and an "end nested top action" right before mutex release. Stasis includes operations that provide nested top actions.

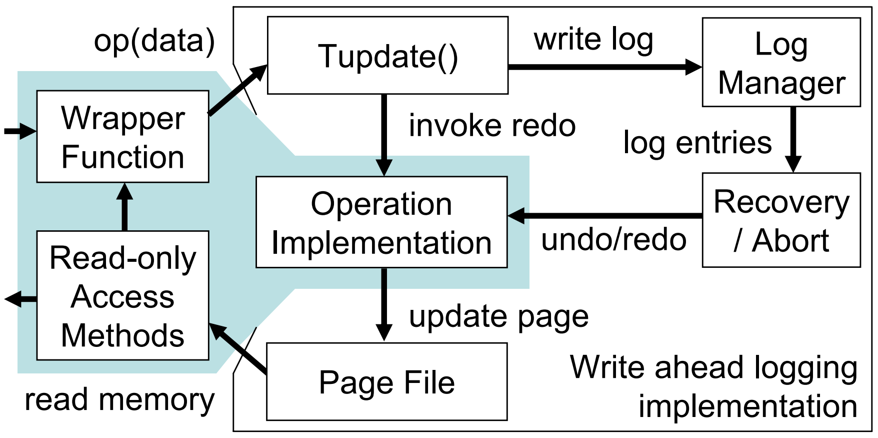


Figure 1.4. The portions of Stasis that directly interact with new operations. Arrows point in the direction of data flow.

If the transaction that encloses a nested top action aborts, the logical undo will *compensate* for the effects of the operation, taking updates from concurrent transactions into account. Using this recipe, it is relatively easy to implement thread-safe concurrent transactions. Therefore, they are used throughout Stasis' data structure implementations. This approach also works with Stasis' segment-based recovery algorithm, described below.

1.2.3 User-defined operations

The first kind of extensibility enabled by Stasis is user-defined operations. Figure 1.4 shows how operations interact with Stasis. A number of default operations come with Stasis. These include operations that allocate and manipulate records, operations that implement hash tables, and a number of methods that add functionality to recovery. Many of the customizations described below are implemented using custom operations.

Operations are invoked by registering a callback (the "operation implementation" in Figure 1.4) with Stasis at startup, and then calling `Tupdate()` to invoke the operation at runtime. Stasis ensures that operations follow the write-ahead logging rules required for Steal/no-Force transactions by controlling the timing and ordering of log and page writes.

The redo log entry consists of the LSN and an argument that will be passed to redo. The undo entry is analogous.[2] Each operation should be deterministic, provide an inverse, and acquire all of its arguments from the argument passed via `Tupdate()`, from the page it updates, or both. The callbacks used during forward operation are also used during recovery. Therefore operations provide a single redo function and a single undo function. There is usually no "do" function, which reduces the amount of recovery-specific code in the system.

The first step in implementing a new operation is to decide upon an external interface, which is typically cleaner than directly calling `Tupdate()` to invoke the operation(s). The externally visible interface is implemented by wrapper functions and read-only access methods. The wrapper function modifies the state of the page file by packaging the information that will be needed for redo/undo into a data format of its choosing. This data structure is passed into `Tupdate()`, which writes a log entry and invokes the redo function.

[2]For efficiency, undo and redo operations are packed into a single log entry. Both must take the same parameters.

The redo function modifies the page file directly (or takes some other action). It is essentially an interpreter for its log entries. Undo works analogously, but is invoked when an operation must be undone.

This pattern applies in many cases. In order to implement a "typical" operation, the operation's implementation must obey a few more invariants:

- Pages should only be updated inside physical redo and undo operation implementations.
- Logical operations may invoke other operations via `Tupdate()`. Recovery does not support logical redo, and physical operation implementations may not invoke `Tupdate()`.
- The page's LSN should be updated to reflect the changes (this is handled by passing the LSN to the page implementation).
- Nested top actions (and logical undo) or "big locks" (total isolation) should be used to manage concurrency.

Although these restrictions are not trivial, they are not a problem in practice. Most read-modify-write actions can be implemented as user-defined operations, including common DBMS optimizations such as increment operations, and many optimizations based on ARIES (71; 96). The power of Stasis is that by following these local restrictions, operations meet the global invariants required by correct, concurrent transactions.

Finally, for some applications, the overhead of logging information for redo or undo may outweigh their benefits. Operations that wish to avoid undo logging can call an API that pins the page until commit (no-Steal), and use an empty undo function. Similarly, forcing a page to be written out on commit avoids redo logging (Force). If the update is to newly allocated space that was freed by a committed transaction then logging may be avoided entirely.

1.2.4 Application-specific locking

The transactions described above provide the "Atomicity" and "Durability" properties of ACID. "Isolation" is typically provided by locking, which is a higher level but compatible layer. "Consistency" is less well defined but comes in part from low-level mutexes that avoid races, and in part from higher-level constructs such as unique key requirements. Stasis and most databases support this by distinguishing between *latches* and *locks*. Latches are provided using OS mutexes, and are held for short periods of time. Stasis' default data structures use latches in a way that does not deadlock. This allows higher-level code to treat Stasis as a conventional reentrant data structure library.

This section describes Stasis' latching protocols and describes two custom lock managers that Stasis' allocation routines use. Applications that want conventional transactional isolation (serializability) can make use of a lock manager or optimistic concurrency control (3; 74). Alternatively, applications may follow the example of Stasis' default data structures, and implement deadlock prevention or other custom lock management schemes.

Note that locking schemes may be layered as long as no legal sequence of calls to the lower level results in deadlock, or the higher level is prepared to handle deadlocks reported by the lower levels.

When Stasis allocates a record, it first calls a region allocator, which allocates contiguous sets of pages, and then it allocates a record on one of those pages. The record allocator and the region allocator each contain custom lock management. The lock management prevents one transaction from reusing storage freed by another, active transaction. If this storage were reused and then the transaction that freed it aborted, then the storage would be double-allocated.

The region allocator, which allocates large chunks infrequently, records the id of the transaction that created a region of free space, and does not coalesce or reuse any storage associated with an active transaction. In contrast, the record allocator is called frequently and must enable locality. It associates a set of pages with each transaction and keeps track of deallocation events, making sure that space on a page is never overbooked. Providing each transaction with a separate pool of free space increases concurrency and locality. This is similar to Hoard (15) and McRT-malloc (63) (Section 2.4.4).

Note that both lock managers have implementations that are tied to the code they service, both implement deadlock avoidance, and both are transparent to higher layers. General-purpose database lock managers provide none of these features, supporting the idea that special-purpose lock managers are a useful abstraction. Locking schemes that interact well with object-oriented programming schemes (116) and exception handling (67) extend these ideas to larger systems.

Although custom locking is important for flexibility, it is largely orthogonal to the concepts described in this paper. We make no assumptions regarding lock managers being used by higher-level code in the remainder of this discussion.

1.3 Transactional segments

The recovery algorithm described above uses LSNs to determine the version number of each page during recovery. This is a common technique. As far as we know, it is used by all database systems that update data in place. Unfortunately, this makes it difficult to map large objects onto pages, as the LSNs break up the object. It is tempting to store the LSNs elsewhere, but then they would not be updated atomically, which defeats their purpose.

This section explains how we can avoid storing LSNs on pages in Stasis without giving up durable transactional updates. The techniques here are similar to those used by RVM (114), a system that supports transactional updates to virtual memory. However, Stasis generalizes the concept, allowing it to coexist with traditional pages and more easily support concurrent transactions.

In the process of removing LSNs from pages, we are able to relax the atomicity assumptions that we make regarding writes to disk. Transactional pages assume that the disk updates each page atomically, regardless of power failures and system crashes. In reality, pages can be *torn* in such situations: an arbitrary subset of a given page's sectors may be written during a crash.

Transactional segments' relaxed assumptions allow recovery to repair torn pages without performing media recovery, and allow arbitrary ranges of the page file to be updated by a single physical operation.

Perhaps more importantly, segments allow log entries and buffer manager updates to be reordered. This allows Stasis to transform synchronous in-memory operations into asynchronous requests. This leads to a runtime decoupling between transactional data structures, the log, and the page file, enabling efficient distributed write-ahead logging implementations.

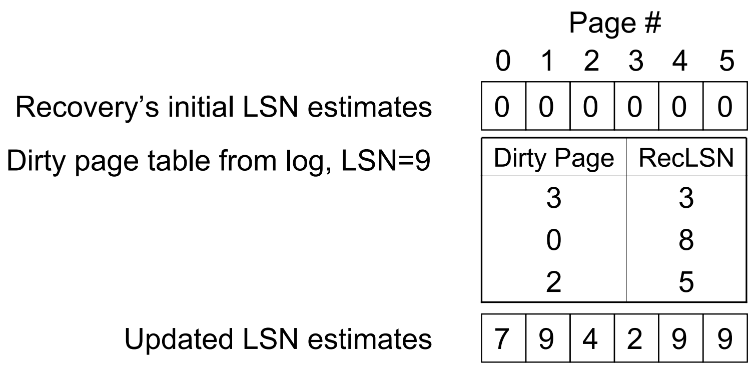


Figure 1.5. LSN estimation. If a page was not mentioned in the log, it must have been up-to-date on disk. RecLSN is the LSN of the entry that caused the page to become dirty. Subtracting one gives us a safe estimate of the page LSN.

1.3.1 Blind writes

Recall that LSNs were introduced to allow recovery to guarantee that each update is applied exactly once. This was necessary because some operations that manipulate pages are not idempotent, or simply make use of state stored in the page.

As described above, Stasis operations may make use of page contents to compute the updated value, and Stasis ensures that each operation is applied exactly once in the right order. The recovery scheme described in this section does not guarantee that operations will be applied exactly once, or even that they will be presented with a self-consistent version of a page during recovery.

Therefore, operations that modify segments must produce deterministic, idempotent redo entries that do not examine page state. We call such operations *blind writes*. A typical blind write would simply write the contents of modified byte ranges to the log. Note that we still allow code that invokes operations to examine the page file, just not during the redo phase of recovery.

Recovery works the same way as before, except that it now estimates page LSNs rather than reading them from pages. One safe estimate is the LSN of the most recent archive or log truncation point. Alternatively, Stasis could occasionally store its *dirty page table* to the log (Figure 1.5). The dirty page table lists all dirty pages and their *recovery LSNs*, and is written to log by ARIES in order to reduce the amount of work that must be performed during REDO.

The recovery LSN (RecLSN) is the LSN of the log entry that caused a clean (up-to-date on disk) page to become dirty. No log entries older than the RecLSN need to be applied to the page during redo. Therefore, redo can safely estimate the page LSN by choosing any number less than RecLSN. If a page is not in the table, redo can use the LSN of the log entry that contains the table, since the page must have been clean when the log entry was produced. Stasis writes the dirty page table to log whether or not segments are in use, so we expect the runtime overhead to be negligible.

Although the mechanism used for recovery is similar, the invariants maintained during recovery have changed. With conventional transactions, if a page in the page file is internally consistent immediately after a crash, then the page will remain internally consistent throughout the recovery process. This is not the case with segment-based recovery. Internal page inconsistencies may be introduced because recovery has no way of knowing the exact version of a page. Therefore, it may overwrite new portions of a page with older data from the log. The page will then contain a mixture of new and old bytes, and any data structures stored on the page may be inconsistent. However, once the redo phase is complete, any old bytes will be overwritten by their most recent values, so the page will return to a self-consistent, up-to-date state.

Undo is unaffected except that any redo records it produces must be blind writes just like normal operation.

1.3.2 Advantages of segments

Because segment-based recovery relaxes the ordering guarantees provided at runtime, it is applicable in a number of environments where traditional page based recovery is known to perform poorly. In particular, segments enable a number of low-level optimizations, such as zero copy I/O, and a significant reduction in memory usage for systems that layer a cache atop Stasis. They also support reordering of log writes and page updates, allowing such requests to be performed in an asynchronous, pipelined fashion. This can be used to coalesce redundant requests, which can save significant amounts of CPU time in object systems. Perhaps more importantly, it allows the requests to be serviced remotely, enabling a new range of distributed storage architectures.

1.4 Large objects and log-structured approaches

In addition to page- and segment-based recovery, Stasis provides a number of mechanisms that support bulk writes of large amounts of data. Although Stasis performs full redo/undo logging by default, it also supports undo-only logging, which force-writes updates to the disk before committing transactions. This is important for large, sequential updates of existing data, as it avoids writing an unnecessary copy of the update to the log. When writing large amounts of data to newly allocated space, Stasis can avoid logging the updates altogether; since the space is newly allocated, there is no need to store the data that it contained before the write began.

These techniques are a natural fit for log-structured systems, which treat the disk as an append-only media. Such approaches avoid logging, and can be easily layer on top of Stasis' large object mechanisms.

Log-structured and append-only data structures are often able to write to disk in a compressed format; the fact that their on-disk representation is append only simplifies this tremendously. Stasis allows such systems to implement custom page layouts, and provides a number of buffer manager callbacks designed to support high-performance compression implementations. These mechanisms have been used to implement *log structured merge trees*, an index structure that provides much higher write and scan throughput than conventional B-Tree indexes. Stasis' log structured merge tree uses a hybrid row-column page layout, and supports a wide range of column database techniques as well as update in place and efficient random lookups.

The primary disadvantage of these approaches is a lack of support for fine-grained transactional updates; instead of committing a few updates at a time, these approaches rely on large, bulk writes to update disk. A number of techniques mask this limitation to some extent, and are presented below.

1.5 Summary of contributions

Stasis is a flexible transactional storage system that is designed to provide high-performance storage to a range of applications, including ones where full transactions are unnecessary. It supports user-defined operations, allowing applications to build their own index structures and disk layouts. By default, Stasis uses a Steal/no-Force storage algorithm that performs full redo/undo logging, but Stasis supports other writeback and logging strategies as well.

Although Stasis is based upon conventional transaction processing algorithms, its architecture is novel: rather than assume a particular data layout or programming model, it leaves such details to the application. This is a departure from database systems and other conventional storage systems, which hard code such details, often according to industry standards or the demands of a segment of the storage market.

Stasis' decoupling of recovery and higher-level primitives led to the creation of a new transactional storage algorithm called segment-based recovery (Section 11.4). Segments improve performance in a number of single-machine scenarios, and recast write-ahead logging as a protocol for building large-scale distributed systems.

Finally, this dissertation presents Rose, a compressed, log structured index that demonstrates Stasis' flexibility (Chapter 10). Rose's data layout and compression techniques borrow ideas from column databases and from log-structured indexes. This allows it to make use of append-only column compression techniques without sacrificing random read performance or the ability to perform random, "in-place" updates. Finally, Rose provides an example of how Stasis can support high-performance, non-transactional workloads.

1.6 Evaluating Stasis

Stasis provides a wide range of storage primitives; some are provided by other systems, while others are novel research prototypes. Most storage systems are, in some sense, interchangeable: they can emulate each other at the cost of application complexity and performance. This section begins with a discussion of Stasis' performance, which is easily quantified, and ends with a qualitative description of our experience building systems atop Stasis and monolithic storage.

1.6.1 Performance

Direct performance comparisons between Stasis and systems such as BerkeleyDB suggest that Stasis is typically 1-3x faster than monolithic systems based on similar index and recovery algorithms. Unsurprisingly, higher-level interfaces such as SQL introduce additional overhead due to request parsing, client-server communication, and so on.

It is worth noting that transactional storage systems are notoriously difficult to benchmark; differences in performance tuning parameters and hardware can easily lead to storage setups that run a few times faster or slower than expected. The fact that Stasis outperforms competitive systems in our micro-benchmarks suggests that its performance is comparable to production systems, not that it will provide uniformly better performance.

However, Stasis does not target applications that are already well-served by existing systems, and in most cases, it does not make sense to reimplement an application from scratch for a 2-3x performance improvement. Instead, Stasis focuses on applications that are a poor fit with existing storage systems, and it is with such applications that Stasis provides significant benefits.

For example, Stasis' segment-based recovery halves the memory required by object storage systems while improving performance. Also, rather than providing a lock manager, Stasis leaves isolation to the application, avoiding the possibility of rollbacks due to transactional deadlock. This greatly simplifies the implementation of highly concurrent systems; with a few hundred concurrent transactions, Stasis provides 10's to 100's of times more throughput than with serial workloads. In contrast, conflicting transactions often cause general-purpose lock managers to degrade performance at such concurrency levels. Since Stasis applications can make use of deadlock avoidance instead of deadlock detection, they can avoid such issues (Section 7.10).

Finally, specialized indexes such as LSM-Trees (Section 10.1) make different performance trade-offs than general-purpose indexes such as B-Trees. Performance models and experiments show that Stasis' LSM-Tree implementation provides orders of magnitude better write and scan throughput than carefully optimized B-Trees, both for extremely large datasets, and for small datasets that fit in memory. For workloads that do not perform random reads, but that do perform random writes, LSM-Trees also scale to working sets hundreds of times larger than can be efficiently supported by B-Trees.

1.6.2 Application complexity

In the process of running these experiments and building a number of other prototypes atop Stasis, we found that interfacing to Stasis is generally comparable to interfacing with other systems; most storage APIs provide access to sets of data, which naturally maps to the index implementations that Stasis provides.

Although Stasis' lack of lock management simplifies many applications, it significantly complicated an application that required conventional lock management but that also needed to make significant changes to lower layers of the storage layer. It may make sense to revisit Stasis' approach to isolation and bundle an optional, general-purpose (index-independent) lock manager for such situations.

Custom index implementations and other low-level primitives can also complicate Stasis applications. Obviously, implementing custom indexes, system catalogs and manually placing data on disk increases application complexity, and increases the likelihood that low-level storage bugs will corrupt data on disk.

Reusing Stasis indexes and isolation schemes addresses these problems; instead of having each application build low-level primitives from scratch, Stasis allows the same set of primitives to be used by multiple systems. We suspect that, to the extent that specialized Stasis extensions provide a better fit than general purpose mechanisms, Stasis will simplify, rather than complicate, application designs. Over the next few years, we hope to make Stasis a practical storage system; hopefully, experience building practical apps atop it will confirm our intuition.

Stasis

Core modules	1220
Allocation	786
Buffer manager	399
Eviction policy	108
Log	595
Total	3108

Transactional data structures

Array	118
Linked list	255
Hashtable	332
Record manipulation	117
Other	900
Total	1722

Page layouts

Slotted pages	297
Fixed pages	108
Other	74
Total	469

Table 1.1. Lines of code in Stasis. Of the 13,000 lines of instrumented code in the Stasis source distribution, approximately 6150 are enabled by default, and another 3400 consist of unit tests. These numbers exclude benchmarking code and Rose.

1.6.3 Stasis' complexity

Most of the complexity of Stasis comes from the interactions that it has with its environment, and not from Stasis' underlying mechanisms. Stasis' implementation is actually quite simple (Table 1.1) and has actually decreased in size over time. The decrease in size is largely due to improvements in Stasis' internal APIs; the code has undergone a number of major revisions that focused upon refactoring code and moving unrelated functionality into separate modules.

Each revision has also brought about new functionality; the first added support for multi-threading, the next for concurrent transactions, then the sequential throughput and compression primitives needed by Rose. The current iteration has two goals: to provide stability and management tools suitable for real world applications, and to introduce a number of concurrency and CPU optimizations that will be necessary as the world transitions to multicore systems and solid state disks.

Stasis' modularity has been crucial to the success of each new revision, and each additional recovery algorithm and class of workloads has benefited Stasis' existing primitives by uncovering bugs, improving performance, and generalizing existing APIs.

1.7 Implementation status and availability

As of this writing, Stasis is still "research-quality" code; it has been used in a number of paper submissions and system prototypes, and works well in those environments. A number of companies have expressed interest in using Stasis in commercial products and work on a "production-ready" release has begun. At the same time, Stasis continues to be a vehicle for storage research; a number of experiments and new extensions are in the works, and are alluded to throughout the following chapters.

Most of the ideas presented here have been implemented, though segment-based recovery is still a work in progress. Stasis is covered by a BSD-style license, and is available at the following address:

`http://stasis.cs.berkeley.edu/`

1.8 Target audience and road map

This dissertation contains a survey of prior work, the experimental basis for Stasis' current design, and places Stasis in perspective with other academic research; hopefully it will be of some use to future storage researchers. It also provides a high-level description of Stasis' API, and a number of examples to help developers make effective use of Stasis and build their own extensions.

The dissertation is divided into three parts. The first is a survey of database data models that have been popular in the past. Relational databases make use of a drastically simplified storage model compared to the other approaches. Although Part I may be of some historical interest, it is intended to be a introductory primer for developers and researchers that plan to build new transactional data structures or to develop data models of their own.

Part II is meant to be an application developer's guide to Stasis. It describes the high-level primitives that Stasis provides, but largely omits the details of their implementation. It proceeds to compare out-of-the-box performance with optimizations based upon the lower-level techniques that are covered in Part III. The second half of Part II explains how developers can build their own transactional data structures atop Stasis. It closes with a description of Rose, a compressed log structured index that exercises most of Stasis' functionality.

In-depth descriptions of Stasis' recovery algorithms and underlying implementation are deferred to Part III. Since the three parts are essentially self-contained, they may be read in any order; readers interested in recovery algorithms or in customizing Stasis (or in working on other transactional storage implementations) may wish to skip to Part III, and eventually return to the other material as needed.

Part I

Survey of database techniques

Chapter 2

Modular database systems

Stasis is a flexible transactional storage system for non-database workloads. Unlike modern database architectures, it does not hard code a particular data model, concurrency control mechanism, or even on-disk layout. Instead, it provides modular implementations of common recovery algorithms and a number of default data structure implementations.

Developers accustomed to modern database development often ask what workloads Stasis targets; after all, database systems seek to provide well-defined, high-level semantics for pre-defined classes of workloads. Stasis takes a different approach, and avoids hard coding workload assumptions. Instead, it provides lower-level primitives than most database systems, allowing application designers to customize Stasis for their needs.

The database assumption is the idea that it is economic to implement a new hard-coded storage system for each new class of workloads, and the assumption that such hard-coded systems will be able to adapt to changes in workloads.

Most databases target similar environments, and therefore provide a similar set of features, such as data independence, data integrity, concurrency control, declarative languages and recovery. Because Stasis leaves assumptions about the workload and deployment environment to the application, it focuses on recovery, and leaves the other mechanisms to higher-level code.

2.1 Design philosophy

As the range of data-intensive applications increases, databases must address an ever-widening range of workloads. Arguably, this approach is now unworkable, as different applications often have contradicting requirements. Proponents of breaking up the database market by workload say that "one size fits all" databases can no longer address the entire market. Although each segment's target workload and underlying assumptions vary, the proposed systems would still be databases; each would assume a detailed understanding of application workloads, then hard- code appropriate mechanisms: "one size fits many."

A number of other approaches adapt database technologies to new workloads. *Database toolkits* emit database server implementations from a set of specifications (Section 2.4.1.1), while *RISC databases* aim to produce better conventional database implementations by combining standardized components. Internally, RISC databases are closely related to Stasis. However, the transactional storage systems most similar to Stasis are minimalist database libraries such as Berkeley DB. Although applicable to a broader range of applications, each of these database designs is based on hard coding workload assumptions and data models. This departs from a cornerstone of software development; software is built from general purpose components.

Database research is increasingly focused upon data management systems created by other communities, leading to some concern within the database community (2). A decade ago, UC Berkeley's database group offered a post-modern database seminar. At the time, it was already clear that the world wide web was fast becoming the global information repository, but still unclear what role databases would play; one goal set forth by the seminar was to "fully realize the potential of the web as a world-wide database system (as opposed to a world-wide heap of mostly unstructured documents)."

A decade later, rather than reshape the Internet into a global database, modern database technologies have adapted and now help enable increasingly sophisticated web services. A similar story played out for large data analysis tasks; MapReduce scaled to web analytics tasks well before analytical databases. It is often said that database technology cannot address web-scale workloads; although this probably is not true, the more interesting result is that the top-down assumption made by database implementers at the time—building for workloads and data models that can be known *a priori*—did not scale to the web.

It is worth noting that web-scale systems have largely been introduced by the operating systems community, which favors *bottom-up* approaches that focus on providing "mechanisms", not "policies". In contrast, the database community traditionally focused on *top-down* approaches that provide well-defined semantics for pre-defined workloads. Both approaches regularly lead to significant advances; it is expedient to examine technical challenges from both perspectives. The first part of this dissertation provides a survey of database technologies with an eye toward physical storage layouts, and concludes (unsurprisingly) that high-level programming and data models should no longer be coupled to the mechanisms used to represent application data.

We often find it expedient to say that Stasis takes a "systems" rather than a "database" approach to transactional storage, or that "Stasis is not a database." Not only are such statements impolite in mixed company, they are imprecise and in some sense misleading; Stasis is essentially a reimplementation of core database algorithms.

The difference between databases and systems built on Stasis comes from the decoupling of storage primitives from higher-level primitives such as data models and transactional isolation. Stasis' design is based on a rejection of the idea that application architectures can *always* be anticipated as the database system is being designed and implemented, or that certain workload assumptions are universal. This leads to a reevaluation of core database design decisions and the architecture described in Part III.

Before jumping into the specifics of Stasis' architecture, Part II explains how to build applications atop Stasis. To some extent, each part is self-contained; one should not need to know how Stasis is implemented in order to extend it with support for new index methods and disk layouts. However, without an understanding of the existing approaches to storage, transactional data structures and data models, developers and researchers frequently reinvent technologies that were invented and carefully optimized decades ago. Part I is a survey of the approaches, from the point of view of data layout and storage primitives.

2.2 Performance and the software hierarchy

It is well-known that most general-purpose storage systems are capable of emulating each other. Therefore, tradeoffs between the various approaches usually are based on one of a few factors: simplicity, performance, and scalability. Although storage designs impact all of these areas, many other mechanisms sit between data-intensive applications and the raw storage interfaces.

If the primitives exposed to the application are inappropriate, then the overall system will behave poorly. In fact, Stasis was originally based on two observations, both of which are symptoms of this problem:

1. When applied to inappropriate workloads, database applications are surprisingly slow and complex: an order of magnitude worse than would be the case if storage hardware and application logic were the bottlenecks.
2. Databases contain general purpose primitives that are unlikely to change across applications or workloads. Databases implement these primitives in a way that prevents reuse.

The layers of software that sit between the storage hardware and the application form a *software hierarchy*. Today, the performance implications of the software hierarchy can be comparable to the performance overheads of accessing storage devices.

Three recent hardware changes have drastically impacted storage designs. The first is the increasing cost of randomly accessing in-memory data. The second is that solid state storage is now practical, drastically reducing the cost of randomly accessing data stored in durable storage devices. The third is that applications now run across many computers in multiple geographic regions, increasing request latencies and failure probabilities. Currently, storage latencies are higher than local network latencies; whether the situation will reverse in a few years is currently a topic of active debate (14; 70; 94; 107). Because of these trends, programming models, system architectures, and storage layouts are evolving rapidly.

Traditional database systems seek to encapsulate and hard code these three aspects of application design. This strategy (the *monolithic approach*) works well in relatively static environments, but leads to complex implementations that are difficult to adapt to rapidly changing hardware and new classes of workloads.

Storage devices are organized into a *hardware hierarchy* that has largely been stable for decades. The hardware hierarchy is organized in layers; the CPUs sit at the roots of the tree (storage hardware is no longer hierarchical), cache and RAM occupy the middle, and for decades, hard disks and perhaps tape drives occupied the leaves.

Within each available technology, storage bandwidth has improved more quickly than latency. This seems to be fundamental, at least for storage technologies with densities that increase exponentially over time. Eventually, this trend leads to storage devices effectively unable to provide random access to data (48). In the past, such technologies (tape, hard drive) were eventually relegated to archival systems, and replaced by newer technologies with superior random access properties. Unfortunately, random access memory is also subject to these trends; for instance, hardware trends suggest that in-memory hash joins, which have poor locality, will soon be outperformed by in-memory sort merge joins, which are based upon sequential operations (68).

Naive approaches to storage issue a single request at a time, paying the latency cost associated with each level of the hierarchy before obtaining or updating useful data. Furthermore, most storage devices are block (rather than byte) addressable; any unneeded bytes in the requested blocks simply waste bandwidth. When coupled with the increasingly sequential hardware primitives mentioned above, this means that the cost of I/O operations, in bytes of wasted bandwidth, has increased over time. Therefore, hardware characteristics have increasingly favored pipelined (asynchronous) sequential requests that keep each level of the storage hierarchy busy moving useful data.

Fortunately, the problem of efficiently laying out and updating data in persistent storage is well understood; a wide range of existing algorithms convert synchronous, random requests into asynchronous sequential I/O. However, changes in the underlying hardware have led to a reevaluation of conventional algorithms and the introduction of a number of new approaches. Stasis' original purpose was to be a platform for this line of research; it avoids hard coding most of the details these systems modify.

However, as changes to storage hardware changed the tradeoffs between persistent data structures, the amount of sequential processing power available to applications leveled off. Current processors contain many cores, but each core provides computational power only comparable to (or even worse than) prior generation cores. Even in systems with ample excess computational power, naive single-threaded software architectures are likely to bottleneck at the CPU or RAM rather than at the storage device.

Although one would expect a carefully tuned database application to bottleneck due to storage hardware (or perhaps database) performance, bottlenecks are now often at interfaces between various software abstractions between applications and storage. These abstractions, or *conceptual mappings*, were introduced for software engineering reasons, and provide the various types of storage and data independence introduced throughout Part I.

For an increasing range of data-intensive applications, traversing these self-imposed mappings introduces orders of magnitude more overhead than the storage hardware. When mappings expose an inappropriate (and hard-coded) interface, it is said that there is an *impedance mismatch* between storage and the application. Hand-generated boilerplate code that deals with such impedance mismatches can easily be more complex than the application logic.

These observations are based on first-hand experience attempting to apply database technologies across a wide range of emerging data-intensive fields. Although databases performed well in a few of these areas, in most cases database-related performance problems or complexity either dominated development time or led to abandonment of the database in favor of other approaches. Furthermore, no single database architecture worked across the handful of applications where databases worked at all. Examination of early surveys and database literature reveals that such issues have existed since the invention of databases.

Figure 2.1 illustrates the performance impact of the traditional software storage hierarchy. Note that, while the figure contains many distinct mappings, modern implementations are built from one or two monolithic components: a database server and perhaps a middleware component. In contrast, Stasis-based applications may operate at any level in the hierarchy; the Stasis performance figures are provided partially for comparison, but primarily because MySQL does not expose comparable primitives.

In summary, Stasis provides each conceptual mapping as a distinct, modular layer, which allows developers to replace monolithic, general-purpose storage software with a mixture of pre-existing Stasis components and custom, application-specific extensions. This allows applications to build (or reuse) custom data structures, isolation mechanisms, and recovery semantics that are well suited

Primitive	1000's of operations per second
Pointer access	5236
`memcpy()`	2358
+ latch	1550
+ pin	815
Stasis' write record	200
Stasis' hashtable update	44
MySQL + InnoDB	11
Random 4KB read from SSD	35
Random 4KB write to SSD	3.5

Table 2.1. The cost of software abstractions is comparable to the cost of disk access. Other than accessing the solid state disk (64), each "operation" consists of updating a 64-bit integer that is already in page cache. The tests saturated CPU, but not disk.

to their workloads. Because storage hardware is currently evolving more rapidly than it has for decades, the need for such low-level control over storage is increasing.

Perhaps more importantly, Stasis allows applications to define custom storage interfaces. With the advent of multicore, and the leveling out of CPU clock speeds, increasingly custom (and increasingly concurrent) storage mechanisms are needed in order to keep up with underlying storage; in fact, all examples in Figure 2.1 were limited by the CPU, not the disk. In practice, most applications access Stasis using lower-level programming APIs than those exposed by database systems. In such systems, Stasis has enabled simpler application designs that provide an order of magnitude more throughput, even before any optimizations due to custom indexing or disk layouts begin to impact performance.

2.3 The Stasis development model

The database industry is structured around a small number of monolithic database implementations. For economic reasons, approximately 5-10 such systems have been able to coexist at any given point in history. To avoid vendor lock-in, customers typically demand that database vendors provide standard APIs, such as SQL or its predecessor, CODASYL. Historically, such standards have been industry-wide, with newer standards either replacing older approaches, or largely failing to supplant the currently popular approach.

In practice, such standards allow developers to migrate between organizations and projects that rely upon databases from different vendors. However, migrating large-scale, mission critical applications from one relational database to another is typically infeasible or immensely expensive. There are a few reasons for this. First, performance characteristics vary between database implementations; queries that are efficient on the legacy system may be hopelessly slow on the new system (either due to missing underlying storage primitives, or due to a poor choice by the query optimizer). Second, data models vary subtly; one database may provide IEEE compliant floating points, while another may not, representations of date values may vary, and so on. Third, different systems have different bugs and are missing different features.

Stasis does not eliminate these problems, but, if used properly, it greatly reduces them for applications that do not need the complexity of a complete relational database implementation. In a well-engineered system, most of these issues would depend on code that is written at a higher-level of abstraction than Stasis provides. Restricting communication between Stasis and higher-level code to a narrow API allows storage and high-level abstractions to be varied independently. Similarly, the mappings above Stasis can be replaced without modifying low-level storage details.

The current, monolithic, approach to storage development has had profound implications for the database industry. Recall Figure 1.3, which illustrates the scale (in number of viable instances) of each level of the modern storage stack. In recent years, application developers have begun to bypass the database entirely; this has led to the "systems" approaches to storage mentioned in Chapter 6.

The problem with these approaches is that they are still largely based upon the monolithic approach taken by databases: they either layer appropriate abstractions atop a general-purpose system, or include hard-coded storage and recovery implementations that were written from scratch.

The goal of Stasis is to allow the industry to support orders of magnitude more custom, durable, storage solutions without giving up the desirable properties of state-of-the-art storage architectures.

2.4 Related work

2.4.1 Database variations

This section discusses database systems with goals similar to ours. Although these projects were successful in many respects, each extends the range of a fixed abstract data model. In contrast, Stasis can support (in theory) any of these models and their extensions.

2.4.1.1 Extensible databases

Genesis is an early database toolkit that was explicitly structured in terms of the physical data models and conceptual mappings described above (12). It allows database implementers to swap out implementations of the components defined by its framework. Like later systems (including Stasis), it supports custom operations.

Subsequent extensible database work builds upon these foundations. The Exodus (24) database toolkit includes the Exodus Storage Manager, which provides low level APIs similar to those of Genesis. It uses abstract data type definitions, access methods and cost models to automatically generate query optimizers and execution engines atop its physical data models.

Object-oriented database systems (76) and relational databases with support for user-definable abstract data types (such as POSTGRES (128)) provide functionality similar to extensible database toolkits. In contrast to database toolkits, which leverage type information as the database server is compiled, object-oriented and object-relational databases allow types to be defined at runtime.

Both approaches extend a fixed high-level data model with new abstract data types. This is of limited use to applications that are not naturally structured in terms of queries over sets.

2.4.1.2 Modular databases

The database community is also aware of this gap. A recent survey (31) enumerates problems that plague users of state-of-the-art database systems. Essentially, it finds that modern databases are too complex to be implemented or understood as a monolithic entity. Instead, they have become unpredictable and unmanageable, preventing them from serving large-scale applications and small devices. Rather than concealing performance issues, SQL's declarative interface prevents developers from diagnosing and correcting underlying problems.

The study suggests that researchers and the industry adopt a highly modular "RISC" database architecture. This architecture would be similar to a database toolkit, but would standardize the interfaces of the toolkit's components. This would allow competition and specialization among module implementers, and distribute the effort required to build a full database (31).

Streaming applications face many of the problems that RISC databases could address. However, it is unclear whether a single interface or conceptual mapping would meet their needs. Based on experiences with their system, the authors of StreamBase argue that "one size fits all" database engines are no longer appropriate. Instead, they argue that the market will "fracture into a collection of independent...engines" (130). This is in contrast to the RISC approach, which attempts to build a database in terms of interchangeable parts.

We agree with the motivations behind RISC databases and StreamBase, and believe they complement each other and Stasis well. However, our goal differs from these systems; we want to support applications that are a poor fit for database systems. As Stasis matures we hope that it will enable a wide range of transactional systems, including improved DBMSs.

To this end, we have begun to adopt the techniques proposed by the RISC database approach. Stasis' implementation is currently unable to make effective use of multicore systems. Although benchmarks of overall system performance provide numbers relevant to end users, they are of little use when attempting to tease out concurrency bottlenecks, or compare multiple implementations of the same component. This is especially true when adding support for multicore systems, as many different Stasis components must take advantage of CPU concurrency before benefits from improving any one component become visible to end users.

Stasis' benchmarking suite now contains a "multicore" directory with a set of benchmarks that espouse the RISC approach. One tests the buffer manager in isolation, while another tests logging, and a third tests the transaction table. A dozen or so such benchmarks are able to cover the majority of Stasis' components. Due to Stasis' modularity, invoking a portion of the system in isolation is straightforward, simplifying the implementations of these tests. Our experience reimplementing Stasis modules for multicore systems has been positive so far; only minor changes to Stasis APIs have been required, and most of the optimizations have been straightforward. Notably, most of this effort has been spent building general-purpose in-memory data structure libraries (such as concurrent hashtables), not modifying Stasis components.

Stasis' use of RISC database techniques extends beyond the benchmarking suite. Wherever possible, Stasis separates mechanisms, such as I/O routines and buffer management, from policies, such as writeback scheduling and page eviction decisions. This has repeatedly allowed us to address Stasis bugs and problematic workloads with minimal effort. Although building a complete database stack atop Stasis is an explicit non-goal of this work, we believe that Stasis is in some sense the first storage implementation compatible with the RISC database approach.

2.4.2 Transactional programming models

Transactional programming environments provide semantic guarantees to the programs they support. To achieve this goal, they provide a single approach to concurrency and transactional storage. Therefore, they are complementary to our work; Stasis provides a substrate that makes it easier to implement such systems.

2.4.2.1 Nested transactions

Nested transactions allow transactions to spawn sub-transactions, forming a tree. *Linear* nesting restricts transactions to a single child. *Closed* nesting rolls children back when the parent aborts (97). *Open* nesting allows children to commit even if the parent aborts.

Closed nesting uses database-style lock managers to allow concurrency within a transaction. It increases fault tolerance by isolating each child transaction from the others, and retrying failed transactions. (MapReduce is similar, but uses language constructs to statically enforce isolation (35).)

Open nesting provides concurrency between transactions. In some respect, nested top actions provide open, linear nesting, as the actions performed inside the nested top action are not rolled back when the parent aborts. (We believe that recent proposals to use open, linear nesting for software transactional memory will lead to a programming style similar to Stasis' (98).) However, logical undo gives the programmer the option to compensate for nested top actions, providing support for closed, linear nesting. Stasis does not currently support arbitrary concurrency models and nestings of subtransactions, since such primitives have not been necessary for Stasis' applications so far. Extending Stasis with support for such primitives would be straightforward, though it would complicate some aspects of the implementation, and perhaps lead to overhead at runtime.

2.4.2.2 Distributed programming models

Nested transactions simplify distributed systems; they isolate failures, manage concurrency, and provide durability. In fact, they were developed as part of Argus, a language for reliable distributed applications. An Argus program consists of guardians, which are essentially objects that encapsulate persistent and atomic data. Accesses to *atomic* data are serializable, while *persistent* data is atomic data that is stored on disk (80).

Originally, Argus only supported limited concurrency via total isolation, but was extended to support high concurrency data structures. Concurrent data structures are stored in non-atomic storage, but are augmented with information in atomic storage. This extra data tracks the status of each item stored in the structure. Conceptually, atomic storage used by a hash table would contain the values "Not present", "Committed" or "Aborted; Old Value = x" for each key in (or missing from) the hash. Before accessing the hash, the operation implementation would consult the appropriate piece of atomic data, and update the non-atomic data if necessary. Because the atomic data is protected by a lock manager, attempts to update the hash table are serializable. Therefore, clever use of atomic storage can be used to provide logical locking.

Efficiently tracking such state is not straightforward. For example, their hash table implementation uses a log structure to track the status of keys that have been touched by active transactions. Also, the hash table is responsible for setting policies regarding granularity and timing of disk writes (140). Stasis operations avoid this complexity by providing logical undos, and by leaving lock management to higher-level code. This separates write-back and concurrency control policies from data structure implementations.

Camelot made a number of important contributions, both in system design, and in algorithms for distributed transactions (38). It leaves locking to application-level code, and updates data in place. (Argus uses shadow copies to provide atomic updates.) Camelot provides two logging modes: physical redo-only (no-steal, no-force) and physical undo/redo (steal, no-force). Because Camelot does not support logical undo, concurrent operations must be implemented similarly to those built with Argus.

Camelot is similar to Stasis in that its low-level C interface is designed to enable multiple higher-level programming models, such as Avalon's C++ interface, which extended C++ with support for distributed, concurrent transactions, and an early version of *RVM*, recoverable virtual memory, which provides transactional access to memory that is managed using special versions of `malloc()` and `free()`.

However, like other distributed programming models, Camelot focuses on a particular class of distributed transactions. Therefore, it hard codes assumptions regarding the structure of nested transactions, consensus algorithms, communication mechanisms, and so on.

More recent transactional programming schemes allow for multiple transaction implementations to cooperate as part of the same distributed transaction. For example, X/Open DTP provides a standard networking protocol that allows multiple transactional systems to be controlled by a single transaction manager (137). Enterprise Java Beans is a standard for developing transactional middleware on top of heterogeneous storage. Its transactions may not be nested. This simplifies its semantics, and leads to many, short transactions, improving concurrency. However, flat transactions are somewhat rigid, and lead to situations where committed transactions have to be manually rolled back by other transactions (121). The Open Multithreaded Transactions model is based on nested transactions, incorporates exception handling, and allows parents to execute concurrently with their children (67).

QuickSilver is a distributed transactional operating system. It provides a transactional IPC mechanism, and allows varying degrees of isolation, both to support legacy code, and to provide an appropriate environment for custom transactional software (59). By providing an environment that allows multiple, independently written, transactional systems to interoperate, QuickSilver would complement Stasis nicely.

The QuickSilver project showed that transactions can meet the demands of most applications, provided that long-running transactions do not exhaust system resources, and that flexible concurrency control policies are available. Nested transactions are particularly useful when a series of program invocations form a larger logical unit (115).

Clouds is an object-oriented, distributed transactional operating system. It uses shared abstract types (116) and per-object atomicity specifications to provide concurrency control among the objects in the system (6). These formalisms could be used during the design of high-concurrency Stasis operations.

2.4.3 Data structure frameworks

Berkeley DB is a transactional storage system quite similar to Stasis, and gives application programmers raw access to transactional data structures such as a single-node B-Tree and hash table (119).

Cluster hash tables provide a scalable, replicated hash table implementation by partitioning the table's buckets across multiple systems (55). Boxwood treats each system in a cluster of machines as a "chunk store," and builds a transactional, fault tolerant B-Tree on top of the chunks that these machines export (84).

Stasis is complementary to Boxwood and cluster hash tables; those systems intelligently compose a set of systems for scalability and fault tolerance. In contrast, Stasis makes it easy to push intelligence into the individual nodes, allowing them to provide primitives that are appropriate for the higher-level service.

2.4.4 Data layout policies

Data layout policies make decisions based upon assumptions about the application. Stasis' data model focuses upon allocation of static length records and regions of disk, and a number of data structures, such as hashtables and growable array types.

This section describes other data layout mechanisms that Stasis could eventually support.

Some large object storage systems allow arbitrary insertion and deletion of bytes (26) within the object, while typical file systems provide append-only allocation (89). Record-oriented allocation, such as in VMS Record Management Services (106) and GFS (43), breaks files into addressable units. Write-optimized file systems lay files out in the order they were written rather than in logically sequential order (112).

Schemes that improve locality among small objects exist as well. Relational databases allow users to specify the order in which tuples will be laid out, and often leave portions of pages unallocated to reduce fragmentation as new records are allocated.

Memory allocation routines such as Hoard (15) and McRT-malloc (63) address this problem by grouping allocated data by thread or transaction, respectively. This increases locality, and reduces contention created by unrelated objects stored in the same location. Stasis' record allocator is based on these ideas.

Allocation of records that must fit within pages and be persisted to disk raises concerns regarding locality and page layouts. Depending on the application, data may be arranged based upon hints (120), pointer values and write order (82), data type (69), or access patterns (147).

Chapter 3

Three eons of database research

The following three chapters span 45 years of database research, and focus upon the interaction of storage primitives and the data models they support. They are based largely upon early textbooks and prior surveys. One such survey spanned the first 25 years of databases (87), with an emphasis on technologies pioneered by IBM. The second spanned the first 35 years (132), and focused upon the evolution of data models, with the hope that later generations of database researchers would read prior work, rather than reinvent CODASYL under the banner of XML.

Similarly, this work is written with an agenda: It documents the storage strategies and data models devised across generations of database systems, and describes the set of primitives each approach can support. Whether or not this paints an idealized picture of early database approaches, it does serve a purpose; it documents state-of-the-art techniques for each of the major approaches to database storage with the hope that this will provide an appropriate starting point for developers of new storage systems. These chapters also document the assumptions upon which the existing systems were built; to the extent that assumptions have changed, revisiting the ideas presented here may be worthwhile.

These chapters are organized according to two significant conflicts within the database community. The first, "The Great Debate," occurred during the transition from *network* and *hierarchic* data models to the relational model. This discussion conflated a number of debates: procedural vs declarative programming, network vs. relational data models, and what this chapter calls *path independent* vs. navigational systems. Because "navigational" is now an overloaded term, this discussion avoids its use. The Great Debate lasted from the introduction of the relational model in 1970 (though its origins can be traced into the mid 1960's), to IBM's announcement that it would transition to relational technologies in 1984.

The second conflict, which is now known in the trade press as the "Vietnam of Computer Science," (102) was touched off in 1985 with the introduction of the term "object-oriented database", and, from the practitioners' point of view, continues to this day. Like the Vietnam War, for various practical and economic reasons, there is no clear "win condition" to the problem of storing objects in databases. The research community (and much of industry) has shifted its focus to other areas, such as providing scalable storage at Internet scale, or optimizing database implementations for in-memory performance.

In early database storage hierarchies, many storage devices backed a single application, and randomly accessing data was prohibitively expensive. With the introduction of direct access storage devices (DASD) this changed to some extent. However, decades later, current hard disks provide essentially sequential access, as improvements in sequential throughput continue to outpace improvements to random access times. This seems likely to change as flash-based storage devices are introduced.

Although storage devices with drastically improved seek performance are beginning to reach the market, RAM's bandwidth has also outpaced its random access times, and high-performance main-memory programs now must be tuned to avoid randomly accessing memory, or otherwise avoid blocking on cache misses. Such techniques are only mentioned in passing in this chapter, and are covered in more detail in Section 10.2.1.

To some extent, storage algorithms are dictated by the ebb and flow of available randomly accessible and sequential storage. Section 6.2 describes a number of Internet-scale systems that make use of storage layouts, and provide semantics similar to those of early hierarchic databases. This should not be surprising, as today's distributed systems bear a stronger resemblance to early database hardware than they do to the hardware that relational systems targeted. Unfortunately, during the transition to the relational model, the community largely forgot the lessons learned during the development of prior generation systems, then reinvented once-commonplace techniques over the following decades.[1] The process of reinventing pre-relational techniques continues to this day; egregious regressions in the state-of-the-art are pointed out throughout the survey, and left for future work.

As the focus of the community transitions away from ACID relational database architectures, next-generation systems are being built without the benefit of the lessons learned from relational (or navigational) systems. Because excellent resources describing the relational approach are readily available, this survey only provides a cursory coverage of relational topics. However, readers unfamiliar with state-of-the-art relational database implementation techniques are encouraged to obtain a copy of a textbook (109; 131; 53) on the subject before attempting to extend Stasis or design a new storage architecture.

Various lines of database research do not impact the storage layer, and instead focus on data models and language semantics. Database systems that focus on these types of innovations are sometimes called $R++$ systems, because they extend the relational model in some way. To the extent that these systems do not impact storage techniques, they are omitted from this discussion. Readers interested in the evolution of query languages and declarative programming will be better off reading the surveys upon which this work is based.

[1]Indeed, much of the work presented here is not available electronically; I have attempted to resolve contradictions in the literature by referencing sources written while hierarchic and network databases were still in use. During this process, the most useful of the early sources has been James Bradley's *File and Data Base Techniques* (19).

Chapter 4

Pre-relational systems

“Navigational” systems (the term was created after the fact) made a number of tradeoffs that varied from later relational approaches. As this chapter describes the evolution of early databases, it presents three such tradeoffs, and argues that they are orthogonal. Although these ideas have been well understood for decades, most non-relational systems are navigational, while most relational systems are not. Today these concepts are often equated, so the idea that the following can be independently varied is still controversial:

- Data models (eg: hierarchic, network, relational, object-oriented)
- Declarativity
- Navigational (path dependent) interfaces

Due to the dominance of non-navigational relational systems, this topic has been largely of academic interest in recent decades.

However, carefully revisiting these concepts leads to a surprising result: the limitations inherent in partition tolerant, distributed databases are analogous to those of navigational systems. Both classes of databases provide *path dependent* semantics; they expose physical cursors to high-level code, and provide query semantics that vary depending on the position of these cursors.

4.1 The master file

The term “data base” was coined in 1964, when a group of workers in military information systems decided to consolidate information across groups (87). Up until that time, each department (application) had its own “master file,” which used its own incompatible format.

Hardware was different at the time; timesharing was a relatively new invention. Commercially available systems did not support DASD (direct access storage devices: hard disks, drums, NVRAM, flash, etc.), and instead relied on punch card or tape. In order to perform “data consolidation” (a join)(87), they resorted to “batch-random” jobs, which were essentially MapReduce tasks, except that the shuffle phase consisted of reading and writing data from multiple tape drives in parallel. Section 10.5 describes a novel combination of standard optimizations from batch-random systems to current, randomly-accessible data structures.

Moving to a single file format allowed the database to be reused across departments, but required applications to use a common set of standard libraries and utilities; a "data base management system," so to speak. This immediately created a number of new requirements: data *consolidation*, *independence*, and *protection* (87). Thus, the field of database research was born.

4.1.1 Data consolidation

Moving data from master files into a standardized format allowed programs (and departments) to make use of each others' data. Without the ability to associate information from one department with information from another, this would have been of limited use; members of each department could simply have learned to use the other departments' software. Therefore, mechanisms that associated pieces of information from each source were needed.

Entity Association uses stored connections between pieces of information to determine whether or not they are related in some way. Over time, mechanisms for entity association have become more complex; initially, only $1 : n$ relationships were supported, then $n : m$ and eventually support for multi-way associations became available.

Entity Matching is the process of inferring connections between existing data, which may include misspellings, abbreviations, and other inconsistencies. Such topics are out of the scope of this survey.

4.1.2 Data independence

Data independence is the idea that data exists independently of its low-level representation, the applications that process it, and even of the database schema.

Application independence is the idea that many programs are able to access the same underlying data. This is perhaps the earliest form of data independence.

Schema independence allows a database schema to be modified without breaking existing applications. Adding new fields is the simplest case, and can be handled in two ways: Applications can be structured to ignore unknown fields, or the database can present a *view* or *subschema* of the database that does not include the new fields. Views and subschemas may be materialized by using *triggers*, or they may be computed on the fly by using *stored procedures.* Both mechanisms are discussed below.

4.1.3 Data protection

For security reasons, different departments (and people within each department) have permission to access to different pieces of information.

Data integrity Bugs in one application should not corrupt data managed by another application. In particular, memory errors should not propagate between applications or to disk, and connections between entities should obey certain integrity constraints.

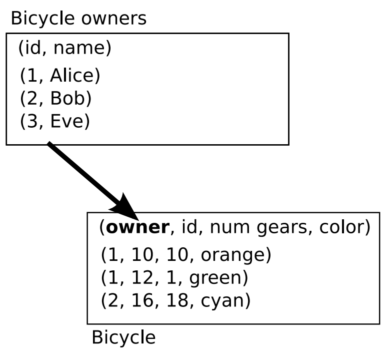


Figure 4.1. A Bachman diagram describing a simple hierarchic database.

Security and access control In addition to improving data independence, views can support a wide variety of security policies. By granting an application access to a view, but not the underlying data, an administrator can prevent an application from accessing sensitive information at the level of fields (columns), or entities (tuples).

4.2 Flat files

In the earliest database systems, each chunk of underlying storage was called a file, and each file stored a single type of record. Access methods corresponded to the features exposed by the tape drive, and there was no direct support for entity association; to associate records in two files one would repeatedly sort the files into convenient orders, then merge the results. This approach provides little data protection beyond using the operating system to set input files read-only. Data independence was also left largely unaddressed, though early report generation languages improved the situation substantially, as did later Unix scripting languages (such as awk, sed and perl) (125).

4.3 Hierarchic databases

The first generation of commercial databases targeted machines without DASD devices. Performance dictated that queries be answered by a single index probe, and economics demanded that the database be *partitioned* across many, transiently available storage devices. Reliability requirements dictated that these systems support *replication* of underlying data.

Such databases were largely limited to $1 : n$ associations between entities; each record had a link to a single parent. The Bachman Diagram in Figure 4.1 describes a simple hierarchic schema, while Figures 4.2 and 4.4 describe two physical layouts of the same conceptual data.

4.3.1 Union types

Initially, each level of a hierarchic database had a fixed type, but it soon became apparent that this scheme was too restrictive. Programming languages include support for *union types* (also called subtyping or inheritance). In a union type, part of a record (struct/object in current systems) may be of some fixed type, while the rest is of one of a collection of types.

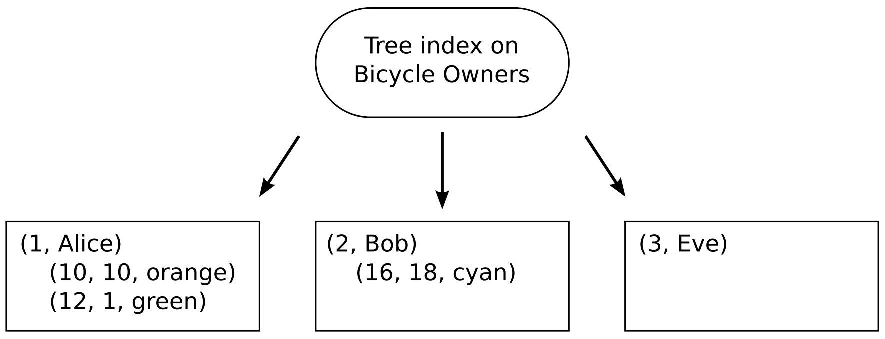


Figure 4.2. A contiguous layout of the hierarchic database. the tuples within each box are collocated on disk. This style of layout is used in today's key-value stores.

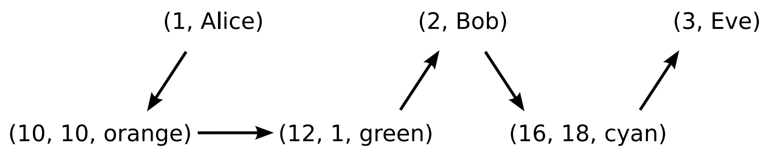


Figure 4.3. A hierarchical pointer layout of the database. The arrows trace a depth first search path. Iterating over the pointers is analogous to scanning an XML document. The pointers decouple relationships between fields from their location on disk, making it easy to update trees of records in place.

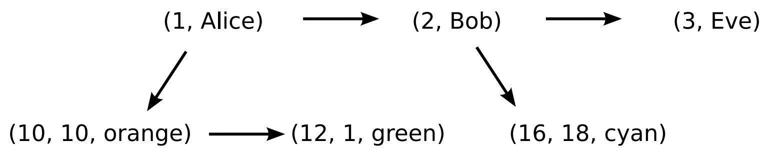


Figure 4.4. A child-twin pointer layout of the database. The twin pointers allow code that traverses the data to avoid materializing the children of irrelevant nodes. However, this layout complicates iteration over the data.

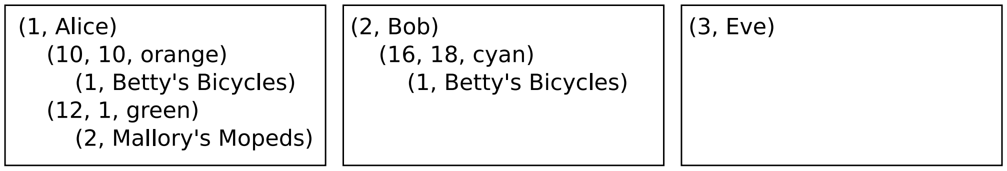


Figure 4.5. Representing non-hierarchic data by duplicating information.

The solution to the problem was simple; hierarchic databases were extended to allow multiple types at each level of the hierarchy.

4.3.2 1:n associations

It is possible to support hierarchic associations without the use of indexes. Conceptually, hierarchic systems store a linked list of records in a pre-order depth first traversal of the tree (Figure 4.3). This is called the *hierarchical pointers* layout. Alternatively, each node could have a list of pointers to its subrecords (*children*), and to the next record that shares its parent (*twin*). This is called the *child-twin pointers* layout (Figure 4.4).

Storing each member of a hierarchy separately in this manner is a *pointer based* approach. Another common approach is to omit the pointers and use *physical contiguity* to group records. This conserves space and improves the performance of read operations, but makes it difficult to add fields to a hierarchy after it has been laid out on disk.

Without indexes, randomly accessing the database requires a scan. Therefore, IMS (the leading hierarchic database) added support for two index structures: hashes (DAM; direct access method) and trees (ISAM; indexed sequential access method). Unlike modern indexes, these structures were not rebalanced; once a tree node or hash bucket was filled, new records were appended to a corresponding linked list that could grow without bound.

In principle, this left IMS with 9 possible representations for hierarchic databases. Any one of the three indexing techniques (none, hash and tree) could be combined with the three record layouts (contiguous, hierarchical, and child-twin). In practice, only certain combinations were supported.

4.3.3 n:m associations and data redundancy

As described, hierarchic databases cannot support $n : m$ associations; each item in the database has exactly one parent. Consider an extension to the bicycle database that modeled warranty service.

If we store the owner of the bicycle in the parent node, then each bicycle record will duplicate information about the warranty provider. (Figure 4.5) Moving the data representation to the parent leads to an analogous problem.

The solution to the problem is to create multiple hierarchies; one of which is the *physical* or *primary* hierarchy, while the others are *logical* or *secondary*. Figure 4.6 describes a physical hierarchy consisting of two trees, while Figure 4.7 describes the associated logical hierarchy that maintains the second set of links.

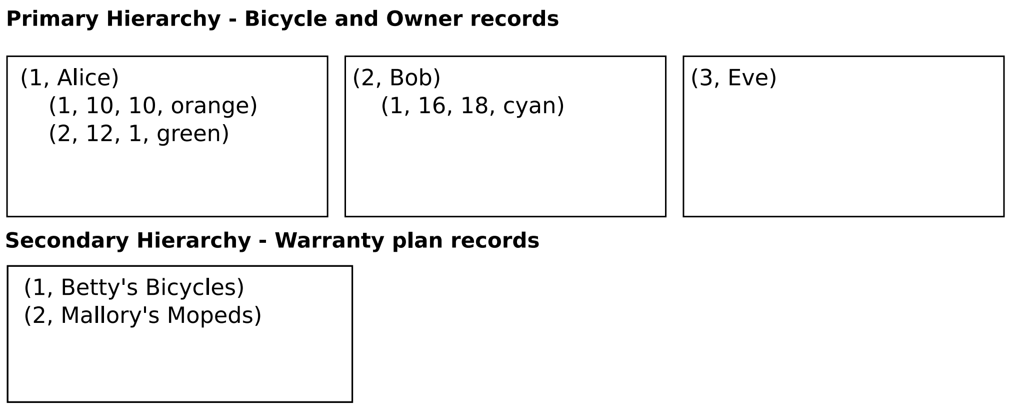


Figure 4.6. A non-redundant layout for non-hierarchic data.

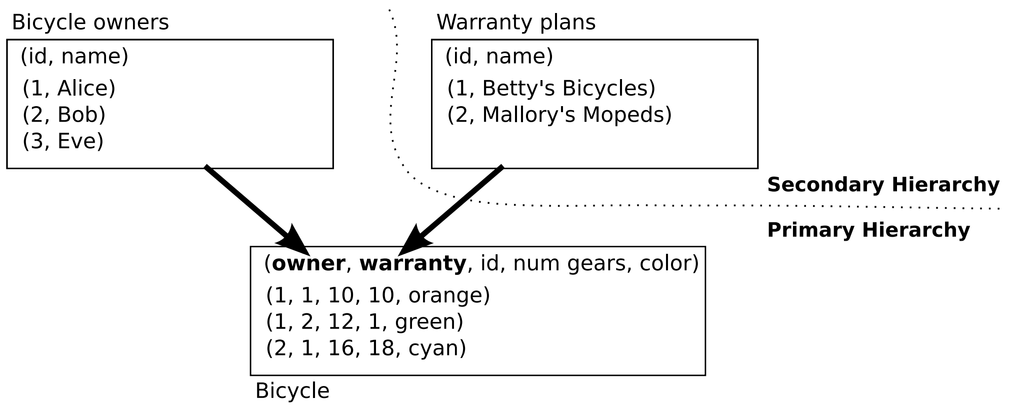


Figure 4.7. Secondary hierarchies add support for non-hierarchic data. Although database designers can arrange for data with a hierarchy to be located together on disk, data from different hierarchies are stored separately.

4.3.4 Data integrity

With the addition of $n : m$ associations came a new class of problems. Up until this point, any set of well-formed hierarchies was a valid database instance; maintaining the physical integrity of the data structures maintained by the database was sufficient. With $n : m$ associations comes the addition of cross-structure pointers ("foreign keys" in the relational approach).

Integrity constraints constrain the set of possible database instances so that invariants regarding the relationships between database entities are preserved. Initially, such constraints ensured that associations reference valid entities, or that they have correct arity; for example, a $1 : 3$ relationship should involve exactly four entities. Over time, integrity constraints were generalized to support increasingly complex relationships.

Note that providing integrity across hierarchies implies that the system is able to atomically update records in multiple hierarchies at a time; in the local case, this requires a transaction. Today. as Internet-scale storage becomes increasingly sophisticated, this problem is being revisited in distributed environments.

4.3.5 Tuple-at-a-time and set-oriented processing

Section 2.2 pointed out that latency due to context switches or network round trips is often the performance bottleneck in modern storage systems. This has been true throughout the history of databases. To address the problem, hierarchic systems introduced the concept of set-based (pipelined) requests.

Consider a program that must compute the mean value of some attribute in the system. A naive approach would request then retrieve each data item in the system, one value at a time. This would incur the cost of a context switch per tuple.

A better approach pipelines the requests and receives batches of tuples at a time. This is difficult to achieve with arbitrary pointer based traversals, as the application logic must examine each data item's pointer before requesting the next item. In order to support pipelined requests (and to simplify application logic), hierarchic and network databases introduced access methods that automatically iterate over the values stored in a database file.

Unlike pointer-based traversals, application logic does not need to process a record returned by the iterator before the database begins to fetch subsequent records. Note that hierarchic pointer layouts readily support iterators that materialize the entire dataset, while child-twin pointer layouts make it difficult to provide general-purpose iterators, but allow applications to traverse the database without processing irrelevant sub-trees.

Iterators amortize the cost of context switches over many data items, but still incur the cost of encoding, transferring and decoding the dataset as it passes between process boundaries. Hierarchic databases expose special purpose languages that allow applications to request tuples, access indexes, update data, and so on. By extending these languages with support for support for iteration (loops), and server-side computation (stored procedures), these systems allowed applications to perform computations against data sets without actually shipping the data across process boundaries.

Recall that these systems were designed for organizations that employed a wide range of data analysts and programmers, with varying security privileges and skill sets. In order to safely move computation from the applications into the database server, these systems provided *memory-safe* languages. Errant application logic written in these languages could not corrupt the database's internal state or violate security policies. This idea can still be seen in modern systems, though general purpose memory safe languages such as Java and C# are beginning to replace custom database languages.

4.3.6 Triggers

Although stored procedures and generic integrity constraints solved a number of issues, they were still unsatisfactory in certain circumstances. *Triggers* are executed on each database update, and can incrementally maintain extra state (such as sums and counts that can be used to calculate mean values), and prevent certain updates from succeeding (which allows them to enforce arbitrarily complicated integrity constraints). Hand-coded triggers can also maintain materialized views, though such code is automatically generated by modern systems (Section 5.5).

Although triggers are powerful, systems that make use of multiple triggers can be extremely difficult to reason about, as it is possible for triggers to run in an unexpected order, and for them to generate tuples that cause additional triggers to be evaluated. Logic languages such as *Datalog* provide an elegant solution to the problem for relational data models (109).

4.3.7 Physical database design and access path selection

Physical database design is the process of producing a database schema and a set of access methods (such as indexes) that accurately represent the data to be stored in the system, while efficiently supporting applications' update and query workloads.

Data in commercial hierarchic databases was accessed via "verb" commands, such as scans, pointer traversals, and lookup by key. Some verbs are *access path independent*, and are guaranteed to behave identically (ignoring performance) regardless of the access paths defined in the schema. However, other verbs (such as pointer traversal) depend on the access methods being maintained by the system. This has the advantage of preventing programs from exhibiting unexpectedly bad performance due to changes in the schema, and the disadvantage of allowing index and other physical database design decisions to break existing programs.

4.4 Network databases (CODASYL)

Although they are flexible, hierarchic data models are unable to represent certain types of data. Similarly, the storage primitives that were used to store hierarchic data made it difficult for programmers to control the placement of certain items, or to choose between various physical representations in a fine-grained fashion. The solution to these problems, and to establishing cross-vendor database standards, was called CODASYL.

CODASYL was based upon a network data model that stored arbitrary graphs of entities and pointers. It essentially subsumed the hierarchic approach, and supported all of the features mentioned above.

4.4.1 Bill of materials processing

At the time, the most important workload left unaddressed by hierarchic databases was *bill of materials applications*, in which a database records the relationships between parts and the (recursive) assemblies that contain them. Such data is not hierarchical, as a part or sub-assembly may be used by more than one assembly. Furthermore, the shape of the hierarchy is not known *a priori*, so the usual approach of grafting a number of hierarchic schemas together was insufficient.

IBM released a "bill of materials processor" (BOMP) data structure that consisted of three (or more) files, each of a single type. The files contained pointers into each other; records with variable length pointers were handled by pointing into a list of contiguous pointers in one of the other files. A bill of materials dataset could be represented by a "parts" file that contained one record per part, and two pointer files; a "contained-in" file, and a "contains-with-count" file. BOMP data structures are able to represent arbitrary network structures.

Eventually, a company called CINCOM released a database system named TOTAL based upon a similar data structure. They extended BOMP with support for multiple record types per file. In 1981, TOTAL was one of the most (if not the most) widely used database systems in the world (87).

Such data structures have enjoyed renewed interest in recent years. Breaking up database entries according to columns makes it possible to execute certain queries extremely efficiently on modern architectures. The modern systems typically make use of per-column compression, and are based upon the relational model.

However, given the historical success these techniques have had in systems based upon graph traversals, introducing support for hierarchic data formats would likely benefit a range of modern applications. Note that the compression techniques used by Rose (Chapter 10) could easily be extended to support hierarchic data.

4.4.2 Owner coupled sets

The main insight behind the network model was that arbitrary databases could be represented using a conceptual model called *owner coupled sets*. An owner coupled set is actually a map that can contain duplicates. The "owner" side of the map is from a single file (allowing support for iteration and set-oriented operations), while the other end of the map could point to any number of files (including the file that contains the owners).

Allowing the set to point to multiple files (and multiple record types) allows owner coupled sets to represent union types. At the time, this feature was only rarely used, though, after the introduction of relational calculus, declarative network languages were introduced, and supported "decompositions" that allowed declarative queries to traverse such sets (18). Unlike current relational systems, this allowed such queries to return records of varying type, which is crucial in object-oriented environments.

Later extensions of relational query languages added support for objects, and provided more complete semantics. However, they were still fundamentally relational; Section 6.1.5 describes a number of performance bottlenecks associated with such approaches. The original description of the network-based approach to union types alluded to these limitations well before the relational techniques were proposed.

4.4.3 Locality optimizations

Allowing records to be associated with multiple master records raises the question of where the dependent records should be located. Network based systems supported at least four different placement policies (ignoring variants due to changing index types or pointer representations):

- With one of the masters.
- Separately from any associated records
- In contiguous chains of associated records
- "Near" the master record

The various layout schemes were partially specified by the schema and partially specified by the application logic that updated the database.

Storing records separately corresponds to the approach taken by relational systems, and by pointer based layouts in hierarchic systems. Storing records in chains corresponds to contiguous record layouts in hierarchic systems. When coupled with indexes and application-managed pointers between records, combinations of these approaches span a wide range of potential layouts.

Although this enabled a wide range of optimizations, it could also make it extremely difficult to write application logic that correctly queried, updated and bulk-loaded the contents of the database. Therefore, the use of such features was discouraged, but left up to the database administrator.

Chapter 5

The relational model

Although network databases are significantly more flexible (and potentially more performant) than relational databases, they provided an extremely complex programming model.

As CODASYL was being standardized, organizations were spending increasing fractions of their development budgets attempting to maintain code written against significantly simpler hierarchic database models. In hindsight, it is obvious that the complexity added by the CODASYL data model would only exacerbate such problems. With this in mind, IBM experimented with storing arbitrary data in purely tabular formats starting in 1965.

However, it was not until Codd introduced the relational model in 1969 that the potential of this approach became clear. Codd provided two sets of operations, relational algebra and relational calculus. He then showed that these operators can compute any entity association expressible with first order predicate logic over arbitrary sets of relations.

5.1 Access path independence

Crucially, both sets of operations are independent of database access methods, ensuring that they will execute even if the underlying indexes and storage layout are modified. This was already true for some hierarchic and network databases deployments, as it was straightforward to support most databases without the use of access-path dependent operators. However, removing the possibility that such operations would be used simplified database server designs, and reduced the cost of maintaining legacy queries. Earlier systems made the opposite tradeoff, and guaranteed that queries would be either executed in a reasonably efficient manner or fail to run. Although the relational model does not provide direct support for this, it is possible to terminate long running queries before they complete. Bounded request latency is becoming increasingly important as the complexity of web service implementations increases, and this problem is now being revisited, albeit with the benefit of new approaches to statistical modeling and declarative programming (7).

5.2 Declarative programming

A related innovation of the relational model was the mapping from relational calculus, which specifies query requests in terms of Boolean formulas over sets of data, and relational algebra, which specifies a query execution plan that performs entity association computations over the database. The three most important relational algebra operators are:

Join A join is the cross product of two sets; each pair of tuples from the two sets will be concatenated, and added to the output set. A *natural join* only emits tuples that have matching values in some columns. This uses a key relationship to associate entities.

Select A selection emits tuples that match some Boolean expression called a *selection predicate*. This computes associations between entities passed as arguments to the selection predicate with tuples in its input.

Project Projection also computes entity associations, though in a more subtle way. Projection eliminates columns from a relation, then collapses tuples that are now identical into a single value, allowing it to simulate entity association calculations based on sort order and grouping. The addition of aggregation functions allows statistics to be computed over sets of associated entities.

When combined with set difference and union, these three operators provide entity association calculations that can be used to compute the results of arbitrary relational calculus expressions. Furthermore, the operators are closed: each takes relations as inputs and returns a relation as output. In terms of conceptual models and practical implementations, this is vastly simpler than the CODASYL approach.

5.3 Late binding

5.3.1 Entity associations

Although relational systems allow programmers to specify integrity constraints, and key relationships emulate pointers, the relational model eliminates pointer-based entity associations. This means that queries can make use of entity associations not anticipated by the schema designer, such as "all employees that make more than George Akerlof and received a raise this year." Like access path independence, network and hierarchic systems were able to support late binding of entity associations; the innovation here is that *all* entity associations can be bound after the query is written. In the network model, it was possible to omit associations from the conceptual model by storing them in pointers rather than record fields.

5.3.2 Access path selection

Recall that access path selection for network and hierarchic systems occurs when the schema is being designed, and when queries are being written; the physical schema is designed to support a set of operators chosen when the queries were written. In relational database systems, queries are specified in relational calculus expressions that map to a wide range of equivalent relational algebra expressions (query plans). The choice of an appropriate query plan can be deferred until it is time to execute the query, and therefore can benefit from statistics regarding available memory and the contents of the relations the query will make use of (118; 135). Again, this leads to a conceptually simpler model for database designers; furthermore it decouples low-level optimizations (such as deciding among join orderings and access methods) from the application, allowing a single query optimizer to improve the performance of many applications. Since optimization is performed at runtime, it also provides some degree of adaptability to workloads that change over time.

5.4 Relational query evaluation techniques

5.4.1 Select

Although arbitrary selection predicates can be evaluated by scanning the contents of a relation, common predicates such as equality, less than and greater than can be translated into index operations. Hash indexes are typically more efficient than tree indexes, but only support equality queries. B-Trees are the most common tree-structured index, and support range queries over a single dimension. Object relational databases (which have little to do with the object databases and object relational mappings discussed below) extended trees with support for more complex predicates such as bounding boxes and "similarity" metrics (32). GiST trees provide a general-purpose framework for implementing trees that support such predicates (60).

5.4.2 Join

Join is potentially the most expensive of the relational operators, as it is the only one that can produce more tuples than are contained in its input. There are three common approaches:

Nested loop join iterates over each tuple of the inner relation once for each tuple of the outer relation

Index join performs an index lookup on the inner relation for each tuple in the outer relation.

Sort-merge join sorts both relations on some key, then scans the two relations in parallel, outputting tuples with matching keys.

The latter two approaches require a selection predicate. For index join, the selection predicate (such as equality or comparison) must map to an index operation. For sort-merge join, the predicate is typically an equality or range computation.

5.4.3 Project

Early relational systems stored records contiguously and made use of indexes such as B-Trees. This left them with little latitude to optimize projection operations. Eventually, new storage models were developed to optimize for queries that made frequent use of projection.

Materialized views are the oldest approach, and significantly predate the relational model. Recall that early database systems also used views to enforce security policies, including restricting access to certain fields.

Datacubes support roll-up queries; the idea is that each column of a database table corresponds to a dimension in a high-dimensional space. Aggregating the values across one dimension (projecting away a column) decreases the size of the data set, and allows potentially interesting queries to be evaluated (such as "provide monthly sales in the state of California"). Aggregating again provides coarser data (year to date sales for California) that can be evaluated using less expensive operations.

Column stores are more recent and optimize for projection queries by storing each column separately on disk. This provides extremely good scan performance since the columns can be easily compressed, and the code that processes them can be optimized to avoid cache misses and branch mispredictions. However, updating data in place in such systems is often infeasible, and randomly accessing a single record can require multiple index lookups, or even multiple scans.

5.5 View maintenance and aggregation

Recall that view maintenance in network and hierarchic systems was based on hand-coded trigger logic. In relational systems, views are usually defined in terms of relational calculus expressions, allowing the trigger logic to be automatically generated. Similarly, aggregation (such as computing the mean value of a field) was implemented either by writing a stored procedure that scanned the table at query time, or by using triggers to incrementally maintain the aggregate's value. Relational systems automate the generation of this code (49), based on various properties of the aggregate function, such as the amount of work that must be performed when a value is added or removed from the relation. For example, to maintain the value of the mean, one would simply track the sum and count of the values. Other aggregates, such as max and median are more expensive to maintain.

5.6 The great debate

Due partially to their simplicity and various technical advantages, and partially due to market forces, databases based on the relational model eventually replaced network and hierarchic databases. However, certain operations are significantly more expensive in relational systems, and the relational approach is subject to a number of scalability bottlenecks. Neither of these limitations was widely appreciated as the industry transitioned to the relational model.

5.6.1 Advantages of the relational model

Due largely to its simplicity, the relational model ushered in a new era of advancements in declarative programming and storage techniques. Some of these advancements were covered above, though the storage techniques will be covered in more detail in later chapters. Because relational systems can be supported by a small number of primitives, these primitives have been carefully optimized over time. B-Tree indexes are a prime example, especially when concurrency control (79; 92) is considered. Furthermore, the relational approach has largely stood the test of time, and has been applied to a broad range of workloads and hardware platforms.

5.6.2 Advantages of navigational databases

The term “navigational database” was invented after the relational model was developed, and is now overloaded. However, the systems that preceded the relational model had a number of features in common that were never really replicated in a relational context. The following issues were well understood well before the relational model was adopted:

Pointer-based join processing Relational systems typically must incur the overhead of an index operation (or worse) in order to join two tuples. In pointer based systems, the disk offset of the data to be joined is stored instead of a foreign key, allowing the join to be performed directly. The primary drawback of this approach is that it makes it difficult to relocate data on disk: all pointers to the relocated item must be updated, or a “forwarding pointer” must be left at the original location.

Schema-independent locality Network and hierarchic systems provided mechanisms to store associated data of separate types inside the same page. An analogous relational approach would allow tuples from different tables to be collocated on disk.

Note that these limitations are not fundamental to declarative query languages or even to the relational model. In fact, declarative navigational languages were introduced after the relational model was well-understood (19), and late network databases were capable of simulating relational databases using storage primitives that supported these optimizations.

Chapter 6

Limitations of relational databases

This chapter describes two fundamental limitations of the relational approach. Although they were not well-understood at the time, they continue to plague certain classes of systems based on the relational model. Unlike many of the critiques of relational systems brought up during the Great Debate, these issues are fundamental to the relational model.

6.1 The "Vietnam" of computer science

As the relational model increased in popularity, so did object-oriented programming languages. Like databases, object oriented techniques are intended to address large, long lived software development projects that involve many developers. Object oriented languages encapsulate data into objects, making it natural to store objects directly in the database. Unlike tuples in a relation, objects of the same type may have different fields.

6.1.1 Union types

Recall that owner coupled sets may contain items of multiple types; iterating over such a set yields a set of records of types $\{T_1, T_2, T_3, ...\}$. Consider a set of records that are "owners" in two such owner coupled sets. For simplicity, assume each set represents a $1:1$ relationship, and each contains only two types of dependent records, $\{S_1, S_2\}$, and $\{T_1, T_2\}$; iterating over the owner records yields pairs of dependent records of types $\{S_1 \text{x} T_1, S_1 \text{x} T_2, S_2 \text{x} T_1, S_2 \text{x} T_2\}$. In general, if n such sets are involved in an iteration, and each set has m types, then the output of the iterator will contain m^n types of records. A relation may only contain one type of record (tuple), and relational operators output relations. Therefore, in order to emulate such an iterator, one would require m^n relational queries. This problem was understood by the end of the 1970's (18), though even proponents of the navigational model treated it as an obscure corner case (19).

6.1.2 Objects: The successor to relations

Object-oriented programming was seen as a successor to prior generation languages (such as "4th generation" database query languages), and the object model was seen as the natural successor to the relational model. To some extent, the technical side of the debate boiled down to two properties of database systems:

Impedance mismatch refers to the overhead (in performance, complexity and developer time) associated with mapping data between the data model used by the database, and the data model used by the programming language.

Language independence is the idea that data stored in a database is accessible, regardless of the language used to implement the application program. Object-oriented databases solved the impedance mismatch problem by giving up language independence.

Impedance mismatch refers to a range of problems, such as varying primitive types (e.g., date formats). However, when storing objects in relational databases, the most significant problem stems from inheritance, where some number of classes have different fields, and "inherit" from some other number (usually 1) of common "superclasses." This problem is exactly the same as the union type problem, bringing a once-obscure corner case in the debate over database models to the forefront of real-world application designs.

Although the union type problem can lead to exponential blowup in the worst case, the best case's increase in complexity is linear (132). In general, for an exponential blowup to occur, one must attempt to iterate over a class that makes use of multiple inheritance, or to perform a join across a set of classes, where each class has multiple subtypes. Still, storing and querying objects involved in complex inheritance hierarchies requires incredibly complex SQL; manually generating such code is tedious (linear in the number of relational operators) at best, and infeasible (exponential) at worst.

Before continuing with the technical discussion, it is important to note that economic factors played an important role in this area. A number of market players chose niches too small to support a monolithic database, while others apparently used techniques such as standardization processes to sabotage technologies as they went to market (132). Earlier surveys cover the development of the database industry in more detail, while the trade press provides a surprisingly detailed analogy between the market forces that played out within the database industry and the events surrounding the Vietnam War (102). There is a growing consensus that, after 25 years of partial solutions, object data models and the relational approach are still mutually exclusive in practice (9). This is *not* to say that no technological solutions exist; the problem is that no practical object database system provides the standard set of database niceties, such as language and data independence, while efficiently and transparently mapping to live, in-memory objects.

6.1.3 Object databases

Rather than build upon the relational model, most object-oriented databases provide a set of primitives that efficiently map to object-oriented programming languages. The most interesting of these (from a storage perspective) is pointer traversal. Object databases typically avoided support for fine-grained, concurrent transactions, and instead focused on workloads such as CAD, which consist of loading large amounts of data, manipulating a significant fraction it, then writing back the data. In such environments, version control is typically more appropriate than transactions.

These systems stored data so that objects were contiguous, and laid out in a similar format to that used by the programming language. Pointers were treated in a number of ways. The simplest approach relied upon 64-bit processor architectures, and maintained a 1:1 mapping between virtual addresses and disk offsets.

This is not a general purpose solution, as today's 64-bit architectures only provide 48 bits (256 terabytes) of addressable space. Although this is significantly larger than current storage devices, capacities continue to increase. Furthermore, hard coding virtual address spaces can make it difficult for a single operating system process to load data from multiple databases.

A technique called *pointer swizzling* was developed to address these limitations(144). Pointer swizzling can be performed eagerly, as data is read from and written to disk, or it can be performed on-demand, as pointers are dereferenced. It is also possible to store frequently used data on disk in "pre-swizzled" formats to avoid the cost of mapping back and forth at runtime. These design tradeoffs can be implemented in a number of different ways. One set of approaches uses the CPU to examine pointers so that they can be swizzled before they are dereferenced. The other set simply attempts to dereference pointers, leading to segmentation faults when a pointer to a disk location is dereferenced. The page fault leads to an operating system trap, and a signal handler swizzles the pointer (and perhaps others in the same page or object). The relative performance of the two approaches depends on hardware characteristics, and the relative frequency of page faults and swizzle checks. Pointer swizzling and the performance tradeoffs surrounding it are well understood (144).

6.1.4 GEM and the R++ languages

Since the relational model became popular, a wide range of extensions to it have been proposed. The vast majority of these are implemented by mapping the extended semantics down to relational operators. For this reason, such languages are sometimes called the *R++ approaches.* GEM is an early, influential example of such systems. It extends the relational model with support for null, aggregation, and a variant of union types called *generalizations.* In some sense, the latter feature makes GEM an *object relational mapping* scheme, though its authors refer to it as a semantic database. Also, unlike object relational mappings, GEM is programming language independent.

The original description of GEM's implementation describes one of the three mapping approaches that are discussed in the next section. The GEM authors consciously decided to base their work on relational, rather than network-based storage primitives (138).

The R++ systems performed the same operations as a relational system would, and provided few performance advantages over existing systems. In fact, while languages such as GEM can save users from hand-generating unacceptable amounts of SQL code, they do nothing to reduce corresponding increases in runtime complexity. If anything, this encourages users to express queries that have unacceptably poor performance.

Ultimately, the R++ languages failed to gain significant market share. It is interesting to speculate on how such technologies (and the object mapping systems that followed them) would have evolved with the additional benefit of network-based storage engines, or at least with access to relational variants of earlier storage layouts.

6.1.5 Object-relational mappers

Object relational mapping technologies automatically translate between object oriented data models and the relational model. Unlike the "object relational" approach, object relational mappings represent each field of each class of objects as a column in a database table. Object relational

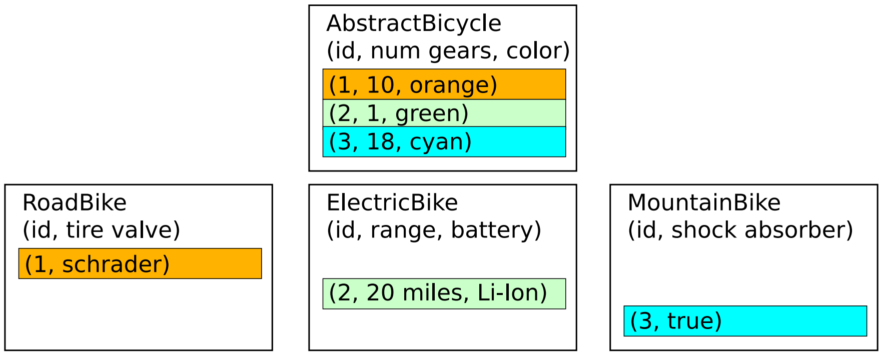


Figure 6.1. Object relational mapping schema with one table per class.

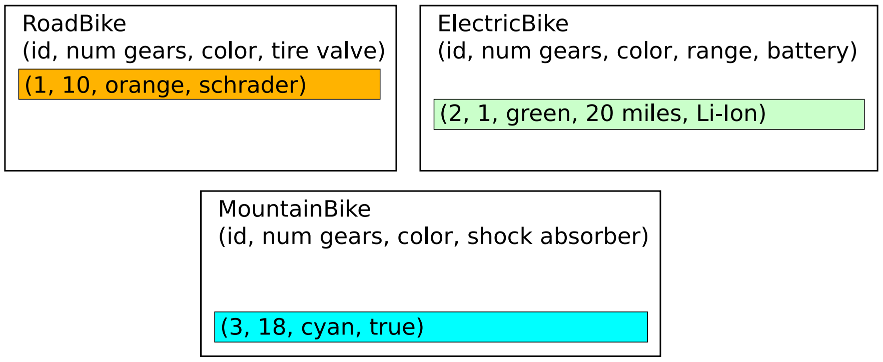

RoadBike
(id, num gears, color, tire valve)
(1, 10, orange, schrader)
ElectricBike
(id, num gears, color, range, battery)
(2, 1, green, 20 miles, Li-Ion)
MountainBike
(id, num gears, color, shock absorber)
(3, 18, cyan, true)

Figure 6.2. Object relational mapping schema with one table per concrete class.

systems store objects as primitive types, so that each object is stored in a single field of a single tuple.

By treating each object as a set of tuples, the queries provided by object relational mappings can make use of the full expressiveness of SQL. In contrast, in object relational systems, each object type must explicitly implement methods in order to expose the contents of the objects to the database system.

Object relational mappings typically express class hierarchies in one of three formats. In the first, each class (including abstract base classes) is associated with a single table (Figure 6.1). Each object is stored as a tuple in a table corresponding to a concrete class. The table is linked with the object's superclass(es) using foreign key relationships. This incurs the cost of a join each time an object is retrieved from the database.

In the second layout, each concrete class is assigned a single table. The declarations of any superclasses' fields are repeated once for each concrete class that inherits from the superclass (Figure 6.2). In order to examine the contents of all instances of some class, one must query one table for each concrete class that inherits from that class.

Both of these two approaches lead directly to blowups mentioned above; each class is represented using a different schema, and so must be retrieved using a different query.

The third layout collapses each class hierarchy into a single table (Figure 6.3). The table contains a column for each field of each class involved in the hierarchy, and fields not associated with a particular class are left null. The storage overhead of this approach is often reasonable

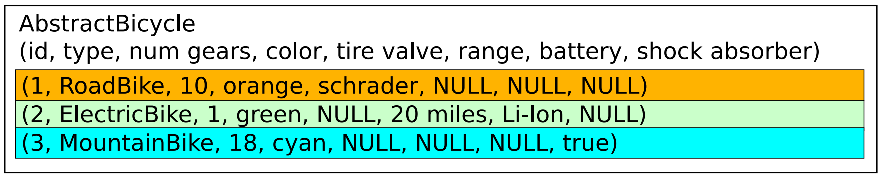

AbstractBicycle
(id, type, num gears, color, tire valve, range, battery, shock absorber)
(1, RoadBike, 10, orange, schrader, NULL, NULL, NULL)
(2, ElectricBike, 1, green, NULL, 20 miles, Li-Ion, NULL)
(3, MountainBike, 18, cyan, NULL, NULL, NULL, true)

Figure 6.3. Object relational mapping schema with one table per object hierarchy.

because modern databases can run length encode null values. Therefore, sparse tables only incur the overhead of a few "null" values and associated counters.

However, flattening all types into a single table leads to a large number of functional dependencies, and omits large amounts of information about inheritance relationships from the schema. With the prior two approaches, constraints on foreign key relationships can ensure that all database instances map to a legal object hierarchy. Once all types are flattened into the same table, such checks are problematic at best.

This approach can return objects of many types using a single scan, since any given tuple may represent any type present in the system. It achieves this by pushing much of the complexity surrounding data types into query predicates, and into the imperative code that processes query results.

A fourth approach, which is not really an object relational mapping, simply stores the object as opaque binary data. Any information that should be exposed to queries (such as keys) can then be duplicated in columns, or exported using the stored procedure mechanisms provided by object relational databases. This essentially reduces the database to a "key-value store" unless database-specific code is added to allow the objects to be indexed and participate in joins.

6.1.6 XML databases

XML is an extensible hierarchic format designed to represent documents. Numerous special-purpose formats have been built upon XML, and a number of those formats have led to special-purpose database implementations.

Perhaps the most familiar extension is used by data sets that represent *ontologies*. Ontological data is essentially hierarchical, except that some nodes contain cross-references to other sub-trees of the database. This is exactly the data model exported by hierarchic databases.

Another extension, RDF, stores entities and relationships explicitly. Each "tuple" in an RDF database contains three columns: subject, predicate and object. This format is particularly useful for metadata information, such as relationships between authors and their works. This approach actually predates XML, and is similar to the "entity-relationship" data model proposed in the early 1980's.

Although the market is still fragmented, current RDF databases are often layered on top of relational systems. Storage of RDF in XML is awkward, as RDF is not hierarchic. Similarly, RDF queries often involve traversals of chains of pointers; translating such queries into relational queries can be inefficient and complex.

A number of general-purpose XML query languages exist. The two most common are XPath and XQuery. XPath is simpler (and is actually a sublanguage of XQuery), while XQuery includes functional language primitives and updates (27). Systems that provide XPath queries over read-only, in-memory databases are fairly mature. However, systems that provide fine-grained updates and XPath queries are still an area of active research.

6.1.7 XML-object-relational mappers

Other than GEM and the XML systems, every technology mentioned in this section is language dependent. However, none of them were seriously supported by language vendors, leading to poorly-integrated, non-standard solutions. In the current (4th) attempt to address object storage in databases, this has changed with the introduction of two mainstream technologies, Ruby on Rails and C#/Visual Basic's LINQ. However, with the rise of XML in the web services space, yet another data model has been added to the mix.

Second generation web technologies ("Web 2.0,") make use of a programming model named AJAX (asynchronous JavaScript and XML) in which bits of JavaScript execute in the web browser, and issue asynchronous requests for pieces of data that are then grafted into the web page's DOM. The DOM, or document object model, is essentially an XML document.

6.1.7.1 Ruby on Rails

Ruby on rails (108) is designed to make it easy to generate SQL code to be run on the database server, and XML/JavaScript code to be run on the web browser. Programs written in Rails must be written using a MVC (model, view, controller (73)) framework, and expose a REST data model (Section 6.2.4) to clients (39). Rails is object oriented, and the code it generates maps between relations, objects and web browser DOMs.

6.1.7.2 LINQ

LINQ abandons the idea of hard-coding tuple or record formats, and instead exposes a set of iterators with various properties. The properties of the iterators are encoded in a type system that is then exposed to the programming language. For a programming language to support LINQ, it must support a number of syntax extensions, and it must make use of an appropriate type system. LINQ can target a number of backend data sources, including SQL, XML, and programming language collection classes. It can also interoperate with event handlers, providing streaming queries that execute callback functions as they generate tuples.

LINQ provides programmers with a data manipulation algebra. In the case of SQL, an equivalent relational calculus expression is then generated from the algebraic expression, and sent to the database server.

6.1.8 Summary of object database interoperability

Although the practical issues associated with these systems have largely been omitted from this discussion, none of these approaches have led to a general-purpose solution to the problem of storing objects in database systems. The XML approaches tend to be used for their original intention: document processing. Rails is essentially language dependent, and focuses on a single (albeit important) application architecture. LINQ focuses on query support and not updates. Also, it addresses a different problem; rather than store language objects in databases, it exposes existing schemas as iterators over structs.

However, this most recent generation of techniques is a significant improvement over the last, mostly due to better language integration. Like the XML document processing languages, the more recent systems have targeted more easily addressable subsets of the original object database problem, and have certainly gained traction.

6.2 Internet scale workloads

Although the relational database approach scales well enough to address many important problems, it fails to support Internet scale workloads because of two bottlenecks.

The first is performance; distributed transactions take longer to complete than local transactions. As systems scale up to larger numbers of users and are migrated to increasingly distributed hardware, the probability that two transactions will conflict with one another increases non-linearly (51).

The second bottleneck is due to the semantics of relational systems. The CAP theorem states that no distributed system can simultaneously provide consistency, high availability, and fault tolerance (44). For a system to be "consistent," it must provide the same answers to queries running on either side of a network partition. In contrast, the answer to a query in an inconsistent system depends on the physical location of the processor(s) executing the query. Such systems violate access path independence, and are in some sense "navigational."

Solutions to these two problems fall into two broad classes. They either avoid providing full database semantics across partitions, or they avoid providing fully declarative programming models. Perhaps unsurprisingly, many such systems make use of mechanisms and provide data models similar to those of pre-relational databases.

6.2.1 Pre-Internet approaches

6.2.1.1 Parallel and distributed databases

Early work in this area fell into two categories. Parallel database systems typically target a single site or organization. They made use of parallelism to provide improved scalability and redundancy to provide availability. This approach scales well to tens, and perhaps hundreds of machines, especially when coupled with expensive, high performance and redundant hardware. However, these systems do not scale well for arbitrary workloads; it is easy to express SQL queries and updates that will lead to distributed deadlocks, poor cache locality, and other issues that will lead to poor scalability.

Unlike parallel databases, distributed databases target multiple organizations or sites, and deal with problems such as access control, resource allocation policies, and heterogeneity between sites (since each portion of a distributed database may be administered by a different team, for different purposes). Distributed databases have been largely superseded by two technologies. The first, transaction coordinators (such as the XA standard (57)), allow multiple servers to coordinate in order to service a single transaction. The second, "software services," provide narrow, standardized interfaces between organizations, allowing fine-grained control over resource utilization and access control between sites. By providing narrow interfaces, software services decouple the underlying infrastructure from the applications that make use of the services.

Both parallel and distributed databases led to a number of important technological breakthroughs, including nested actions (transactions), and protocols such as two-phase commit (38; 97). Most relevant to Stasis are probably the various strategies for distributed joins, table shipping and replication.

6.2.1.2 Heterogeneous storage

Progress in storage hardware has been dominated by two types of events. First, as storage density increases, the bandwidth provided by a given technology also increases. While the absolute latency of random requests also increases, these improvements are outpaced by the increases in bandwidth. The end result has been that, over time, a given storage technology increasingly provides sequential access, not random.

The second type of event occurs when a new class of storage devices is introduced. In the case of durable storage, punch cards were replaced by tape, which was in turn replaced by DASD devices (hard disks and drums), which are now likely to be replaced by flash devices. Tape and hard disk densities (and therefore bandwidth) continue to increase; disks replaced tapes due to latency, not bandwidth, and flash is replacing disk for the same reasons. If newer, byte addressable technologies such as phase change memory replace flash, it will likely be due to request latency, not bandwidth or absolute capacity.

The end result of these trends is that prior generation storage devices provide extremely high capacity at low cost, but that randomly accessing the data they store is infeasible. Therefore, technologies that migrate infrequently (or only sequentially) accessed data to low-cost storage have been introduced at regular intervals.

Recall that hierarchic databases were organized into "files," each of which corresponded to a physical device that could be mounted or unmounted at runtime. The access methods these systems provided were organized in terms of files, minimizing the number of times physical media needed to be manually installed and removed in order to service a request.

A number of newer techniques deal with migration of data between fast random access devices, and slower, or write-once directly accessible devices.

HP AutoRAID (146) migrates pages between expensive, but quickly updated mirrored hard disks (RAID-1), and inexpensive, but slower schemes that make use of parity bits (RAID-5). These techniques worked for data that was updated in place.

POSTGRES' original, no-overwrite storage system made use of algorithms appropriate for write-once data, and provided versioning of the data it stored. Its vacuumer was designed to move old versions of data to write-once devices (127).

Section 10.1 describes LSM, or *log structured merge* trees, which convert random index updates into purely sequential I/O operations. The LHAM extension to LSM-trees used merge processes to migrate old data to write-once, or lower performance media (100).

6.2.2 Early Internet approaches

The storage approaches up to this point target installations with strict data integrity demands, and are designed to run on top of reliable hardware. Of course, "reliable" is a relative term, as all hardware eventually fails. This section talks about techniques that address "unreliable" hardware installations: systems with 100's to 10,000's of machines. Even if each node in such a system is "reliable," it is likely that some nodes will be unavailable at any given time.

6.2.2.1 Search engines

From the storage perspective, the key insight behind search engine architectures is that approximate results are acceptable; it is rare for a user to notice if a number of web sites are missing from the first page of search engine results. Also, some sites are more likely to be missed than others; they tend to be located toward the top of search results, and the keywords they are associated with are frequently requested. Replicating such sites across many machines increases the likelihood that at least one copy of their entry will be on a working machine at any given time.

Early search engines made use of batch updates; a web crawl would be performed periodically (perhaps a few times a month), the search index would be regenerated, and the production machines would migrate queries onto the new index. Over time, the interval between updates has decreased, and has been made variable; important news sites are crawled more often than infrequently updated pages with little traffic.

Instead of B-Trees indexed on a handful of keys, search engines make use of inverted indexes, which provide efficient full-text queries. Detailed information about query execution strategies and result ranking is outside of the scope of this work, and detailed descriptions are readily available (21; 109).

6.2.2.2 Key-value stores

Search engine architectures are geared toward high-latency, bulk updates, and queries that consider large amounts of data. Key-value stores lie on the other end of the spectrum, and provide extremely efficient 1:1 mappings between identifiers and data. Cluster hashtables (41) are perhaps the simplest approach, and simply apply a hash function to each identifier. The result of the hash maps to a node (or set of replicated nodes), and the request is forwarded to the appropriate machine.

Because data is partitioned based on a hash function, each machine is likely to receive an equal amount of load. However, highly skewed access patterns (where the number of machines is comparable to, or greater than the number of popular values) can lead to unevenly distributed workloads. Also, if the size of the data or the number of clients increases, then more machines must be added to the cluster. Similarly, broken and obsolete machines are continuously removed from the cluster.

This means that scalable key-value stores must support addition and removal of machines at runtime. A number of network protocols and policies can be used to support this functionality (56). From the storage point of view, adding and removing nodes presents a number of challenges. Migrating data off of already overloaded machines is likely to be time consuming and disruptive to clients, so its impact should be minimized. This typically means that it should be possible to copy data using purely sequential I/O.

Also, the size of data and expected distribution of requests impacts the choice of index. If data is small and bounded in size, then it can be stored alongside keys. If the data can be extremely large, then it may make sense to store it separately. Similarly, if the system chooses the keys (and not the clients), then it may make sense to store data in tree structures, rather than hashes. By choosing monotonically increasing identifiers, the system can ensure that new data is always appended to the end of the tree, avoiding rebalancing costs.

6.2.2.3 Chunk stores

A third set of workloads involves large data that is infrequently updated, such as map tile images for online mapping services. Such data can be stored in a file system using a simple naming convention, or can be stored as database blobs. A number of issues complicate such systems. Taking maps as an example, information about access locality is known *a priori*; adjacent tiles are likely to be accessed at the same time. Storing them adjacently on disk avoids seeks. On the other hand, storing all the tiles for a popular tourist attraction is likely to cause load imbalances (10).

A second set of issues come into play when such data is periodically updated. If the data placement policy of the underlying storage system is not carefully designed, fragmentation (non-contiguous placement of files) can rapidly degrade system performance (117).

6.2.2.4 Transactional chunk stores

If data stored in a chunk store needs to be updated with low latency or fine granularity then additional mechanisms are required. Boxwood (84) consists of a set of chunk stores (their chunks are variable length and replicated), and logging services that are suitable for B-Tree maintenance. Boxwood focuses on clusters of approximately 8 machines. Sinfonia is a similar approach, but instead provides such primitives using a mechanism called *minitransactions*: ordered, atomic test and set operations over arbitrary memory addresses. Sinfonia provides lower-level building blocks than Boxwood, and focuses on applications that make use of an order of magnitude more machines.

6.2.2.5 MapReduce

MapReduce is a programming model for large scale data analytics tasks. Data is partitioned, sent to mapping tasks, then resorted and shipped to reducing tasks. The sort order ensures that related entities are sent to the same reduce task (35).

The dataflow of a MapReduce task could easily be replicated using a parallel database system. Unlike parallel databases, MapReduce is designed to tolerate multiple node failures per-run; in the worst case, failures cause data to be omitted from the output. Also, MapReduce tasks are not limited to relational formats or operators. Instead, mappers and reducers are written in general purpose programming languages.

From a storage point of view, MapReduce is a substrate for scalable, sequential data manipulation. In this respect, it is simply a scalable system for managing flat files.

6.2.3 Scalable database architectures

Although Internet scale systems blur the line between "databases" and "systems," newer approaches layer higher-level primitives and improved semantics atop earlier approaches. There are currently two popular approaches; the first builds scalable systems atop databases, while the second uses scalable services to implement databases.

From Stasis' point of view, the two approaches are equivalent; the storage layer should be agnostic to the primitives layered on top of it, and ultimately, it is likely that both approaches will require similar storage primitives.

6.2.3.1 Sharded databases

The simplest way to scale up a database workload is to partition the dataset across many database instances. Each such partition is called a *shard*, and is essentially independent of the other partitions. This does nothing to provide transactions across partitions, but it does provide linear scalability. Since such systems cannot perform atomic queries across partitions, and partition boundaries may change over time, applications written atop such systems often treat the database as a simple key-value store, or at least arrange for joins to involve a small amount of state that can easily be split along consistent partition boundaries.

Perhaps because it is difficult to write complex queries atop such systems, databases such as MySQL, which provides extremely fast tuple manipulation, but limited support for query optimization and joins, tend to dominate in this space.

6.2.3.2 Shared-nothing databases

In contrast, shared-nothing databases leverage the high-level semantics of SQL to provide scalability. These approaches are based upon rewriting a single SQL query into a set of queries that can be run against each partition of a sharded database. The results are then joined with each other or otherwise consolidated, providing the user with the illusion of a single database system.

In principle, existing transaction coordinators (which are based upon two-phase commit) could provide fault tolerance to such systems. However, as the number of partitions grows, the likelihood that each database instance is available decreases. Replication of each shard provides better read availability, but increases the costs of updates, and complicates mechanisms that provide consistency. Ultimately, these systems must trade off between consistency, availability and partition tolerance. The loose coupling of the shards provides a more flexible set of tradeoffs than existing, monolithic parallel database implementations.

6.2.3.3 Explicit locality: PNUTS

At the same time as more scalable database implementations were introduced, key-value stores began to provide expanded semantics. PNUTS is a recently example, and provides applications with explicit control over consistency. In particular, it provides primitives such as atomic test and set, which enable ACID updates to a single row at a time.

It also provides support for request ordering; applications decide how fresh the data they access

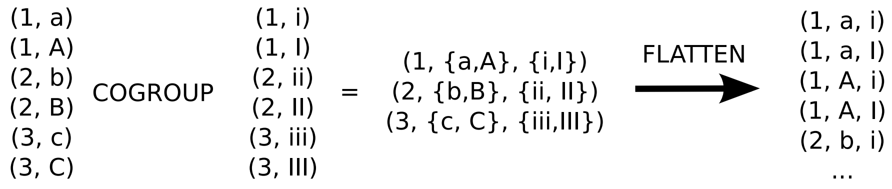


Figure 6.4. Pig's FLATTEN and COGROUP operators use nested tuples to avoid enumerating cross products.

must be. A motivating example for this primitive is access control; if a user denies public read access to a directory, then uploads files, the system should ensure that the files are never made publicly available.

Finally, PNUTS provides applications with control over where data is placed; it is designed to run across multiple geographic regions and applications may migrate the master copy of a particular key-value pair from one region to another. This reduces the cost of updates and the cost of providing consistent reads when the users accessing a given data item are located near each other.

6.2.3.4 Scalable analytical processing for hierarchic databases

Pig provides a high level, but imperative query processing language that can be run atop Hadoop, a MapReduce implementation. It focuses on ease of use for large, semi-structured data sets. Because executing a job is extremely expensive, it does not attempt to optimize queries, and instead trusts the user to structure their queries in ways that will lead to efficient execution. However, conservative optimizers are mentioned as future work. (104)

Its data model is essentially identical to that of a hierarchic database, with two exceptions. First, it supports schemaless hierarchic data. In principle this is no more expressive than conventional hierarchic systems. In practice, it saves users from the burden of writing potentially complex expressions in a data definition language. Second, Pig does not support pointers (where one tuple stores the file offset of a tuple in a separate file), and instead only supports equijoins based on values. This is likely due to the fact that Pig is not designed to provide efficient point lookup queries.

In addition to tweaking the underlying data model, like late hierarchic and navigational designs, Pig's query language is informed by relational algebra and calculus. Pig's designers opted to include primitives analogous to relational algebra, but not calculus, as the cost of an optimizer error outweighs the benefits of declarative programming for their target applications.

From a hierarchic point of view, a relational equijoin is an entity association followed by a denormalization step that introduces functional dependencies into the data. This causes a quadratic blowup in the size of the underlying representation (Figure 6.4). Pig's join operator is split into two separate operators. The first, COGROUP, groups multiple input sets on some number of keys, and provides nested tuples as output. The second, FLATTEN, computes the cross product of the nested sets in each tuple. In addition to lowering the costs of query execution, Pig's users say that this leads to semantically cleaner programs. Like relational algebra operators, each operator in the hierarchic algebra provided by Pig is easily parallelizable.

Unlike early databases, which hard-coded special-purpose languages, Pig follows the lead of recent relational systems, and provides support for user defined functions that allow queries to execute code written in Java (or other standard languages) inside of the query execution pipeline. Pig does not provide a data manipulation language, though it does allow queries to specify the format that will be used to materialize results to disk. Pig also provides a debugging environment, which locates, or fabricates data that exercises Pig queries, allowing programmers to reason about the behavior of their code without waiting for a Pig job to be staged and executed.

A similar system is based upon Dryad, which provides MapReduce style programming primitives over arbitrary directed acyclic graphs (65). Such extensions to the MapReduce execution model are then leveraged by a second system, DryadLINQ, which translates LINQ requests (Section 6.1.7.2) into Dryad jobs (148).

Since the values that LINQ manipulates can be of arbitrary types (tuples, objects, XML), it provides a data model that is more general than Pig, but gives up language independence. In turn, this could lead to interoperability problems between DryadLINQ and other systems. It also makes it difficult to implement a Pig-style debugger for queries that make use of complex data types. Of course, LINQ queries that restrict themselves to simple types, such as Pig's nested tuples, can avoid these issues. LINQ's query model provides a generalized algebra that is based upon a single operator *SELECT-MANY* that subsumes all of the relational operators, and many operators from other algebraic query languages. Interestingly, it has strong theoretical ties to monads, an important primitive in pure functional programming languages (90).

6.2.4 Caching and the REST data model

Recent advancements in web browsers support applications based upon asynchronous calls that update portions of the screen at a time. The *presentation layer* has long been a subject of database research, as it impacts various database design decisions. In the case of Internet applications, moving from static result pages to interactive user interfaces increases the importance of request latency, and of caching.

The solution to these problems is called the "REST" (representational state transfer) data model (39), and really refers to the idea that applications should make appropriate used of transport layers such as HTTP. For an application to be RESTful, its requests must have a number of properties:

- Client-server
- Stateless
- Cacheable
- Uniform interface (Per-object, not per-session, names)
- Layered system (From the clients' point of view, caches and other intermediate nodes are indistinguishable from the server)
- Code on demand (Optional; user interface code is transferred to clients on demand)

These design constraints allow applications to be serviced by incrementally moving requests into a distributed cache. On the simplest end of the spectrum, static objects (such as company logos and style sheets) can be stored in content delivery networks, so that they are serviced by local servers rather than transported across large geographical distances.

Next, "mostly" read-only objects (such as Wikipedia articles) can be pushed into cache and invalidated each time they change. For read-mostly workloads, this moves load from the application servers and database servers into the cache services. Ultimately, this allows applications to explicitly migrate the master copy of frequently written objects to servers closer to the clients that update them (as with PNUTS; Section 6.2.3.3).

In some sense, web-services based on HTTP now run atop a global, distributed (administered by multiple organizations) database system. Unlike conventional database systems, where client machines directly communicate with application logic, which in turn sits atop a cache, web-based systems insert a cache between the clients and the application logic.

6.3 A note on cyclic time

Due to the rise of XML and object databases, and their similarity to the CODASYL model, a prior survey posited that database research moves in cycles marked by the repetition of various data and storage models. It is perhaps more useful to consider a spectrum of possible hardware models, with purely random access primitives on one side, and purely sequential primitives on the other. This gives rise to various combinations of storage models and data models; it causes the technical and economic tradeoffs associated with each approach to vary over time.

It is interesting to note that, as the relative cost of random accesses decreased, the industry moved to a data model that was less focused on providing predictable performance and scalability, and instead focused on providing predictable semantics and improved programmability. Now that this hardware trend has reversed, systems are being designed to make efficient use of increasingly non-uniform memory architectures. Such systems are often based on storage and data models reminiscent of hierarchic and network approaches.

Rather than attempt to provide interfaces appropriate for typical database workloads, Stasis abandons the mechanisms that guarantee data independence, integrity and protection. Furthermore, it is *data model independent* and *storage model independent*; it is designed to efficiently support arbitrary storage models, making it an appropriate (in principle) storage system for any data model that would benefit from transactional storage.

The assumption is that a wide spectrum of systems can be layered on top of the mechanisms Stasis exports. Hopefully, rather than start from scratch, these systems will be informed by decades of database and storage research, allowing them to outpace monolithic approaches, while exporting programming models that provide good performance at appropriate levels of abstraction.

Part II

Writing Stasis Applications

Chapter 7

High-level interface

7.1 The Stasis data model

Stasis is designed to support arbitrary data models. As Chapter 4 explained, network database systems had a similar goal, and it is indeed possible to simulate the other database approaches atop such systems. However, this led to complex database implementations, stylized "cursor-based" programming models, impedance mismatch, and a number of other problems. Ultimately, network databases were replaced by the relational model, which largely leaves these problems unaddressed, except that it leads to simpler, more easily understood systems.

There are two ways to generalize a system: adding functionality and removing it. Unlike CODASYL, Stasis takes the latter approach. It only supports two data types, `recordid` and `record`, and specifies an on-disk format for neither. `records` are simply arrays of bytes, and `recordids` are stored inside of `records`. The only in-built notion of `recordid` in Stasis is an in-memory structure that consists of a page offset, a slot within the page, and a size.

Pages are currently the unit of recovery, though support for variable-length *segments* (Section 11.4) will relax this over time. Slots are logical offsets within pages and are not used by the "core" Stasis library. However, they are used throughput most higher-level data structure implementations, and are included in `recordids` for convenience. Disk layouts that do not make use of slots may simply pass in a dummy value, such as zero, whenever Stasis requires a `recordid`. Stasis provides two primitives atop these types: `Tupdate()`, which logs and updates a page, and `loadPage()`, which provides a pointer to a pinned, in-memory copy of an on-disk page.

It is difficult to work directly on top of these low-level primitives. In fact, much of the time and effort that has gone into Stasis has been dedicated to building higher-level primitives such as indexes and allocators. However, each piece of these higher-level mechanisms has been designed with modularity in mind. The result is a library of data structures and other storage management primitives that can be easily extended, reused, or replaced with better implementations.

This chapter describes Stasis' high-level data structures in a way that is meant to be accessible to application developers. The following chapters examine the underlying primitives, eventually grounding the system in terms of buffer manager and log operations. In turn, Part III explains how Stasis' recovery algorithms work, and how the log and buffer manager work together to support the primitives described here.

7.2 Bootstrapping a database

The first problem one is likely to encounter when using Stasis is the problem of writing a program that creates a database the first time it is invoked, but that automatically reopens the database on successive invocations. One could imagine checking to see if Stasis' backing files exist; if so, the program would infer that the underlying files have been correctly initialized.

However, this leaves a race. It could be that some prior invocation of the program crashed after the files were created, but before initialization completed. Besides, it is possible that the location or format of Stasis' backing files will change over time. It would be inappropriate to hard-code assumptions about the underlying I/O mechanisms in application code.

The solution (perhaps unsurprisingly) is to use a transaction. Stasis provides a `record` allocation function, `Talloc()`, that will allocate a `record` of some requested size and return a `recordid` that points to it. `Talloc()` is guaranteed to return a `recordid` that points to the zero'th slot of the first page the first time it is invoked against an empty database.[1] A second function, `TrecordSize()` returns the size of a `record`, or the value `INVALID_RECORD` if the record does not exist. Therefore, at startup, the application need only start a transaction and invoke `TrecordSize()` on page 1, slot 0. It can then either continue with normal operation, or bootstrap the database and commit the transaction. Stasis does not include a concept of a system catalog, so the application may do whatever it likes in order to bootstrap the database. A common choice is to allocate a hash index that will store `recordids` that point to the rest of the data structures in the system.

7.3 Allocation units

Bootstrapping the database introduced a number of concepts. Of these, allocation is the most fundamental. Stasis' allocator actually consists of three layered allocation mechanisms: the region allocator, the page allocator, and the record allocator. `Talloc()` is the record allocator. Before allocating a record, it must obtain space for the record to occupy. It does this by asking the region allocator for a region of contiguous pages. It then formats the pages in the region as *slotted* pages (Section 9.7.1) so that they can store small, variable length data. Space is then drawn from this pool each time `Talloc()` is invoked.

Note that use of `Talloc()` is optional; applications are free to allocate regions directly, and to populate them as they see fit. The region allocator stores an allocator id with each new region, and provides iterators over the regions stored in the system. This allows application-specific allocators (and `Talloc()`) to determine which regions are under their control. Of course, more sophisticated schemes are possible; the mechanism provided by the region allocator provides a simple default.

In the current implementation, each region incurs significant overhead; in particular, the region contains a 4 kilobyte header page, and will be enumerated on startup as allocators attempt to determine which data they manage. Clearly, this scheme could be replaced with more sophisticated and more efficient approaches, but regions are meant to be much larger than a page, reducing the importance of such optimizations. The *page allocator* bridges the gap, providing efficient allocation

[1] The zero'th page is reserved for Stasis-specific information, such as magic numbers, allocator state, and Stasis version information.

of small numbers of pages at a time, and allowing callers to pass in locality hints, improving the chances that related data will be clustered on disk.[2]

7.4 Records

Records are intended to be the smallest unit of allocation in Stasis. They correspond to a contiguous range of bytes and can be updated using a number of pre-defined mechanisms, including functions that set an entire record, increment integer-sized records, and byte-range operators that update a subrange of a record at a time.

Stasis includes a number of default page layouts, including pages that support records of varying length and pages that can only store records of a single length. These pages implement a standard API, allowing a single "record set" log entry format to be applied to a range of disk layouts.

Since records are typically stored on pages, they are naturally limited in length; normal records cannot span multiple disk pages. For convenience, Stasis' record allocator detects requests to create larger records, and automatically allocates a *blob* (binary large object) record instead. Blob records may span pages.

This allows applications to safely call `Talloc()`, regardless of whether or not the requested record will fit on a single page. Unlike updates to objects that fit within a single page, updating a blob requires multiple log entries. Therefore, special purpose operators (such as "set record range") must explicitly check for blobs and provide logic to handle blob updates. Therefore, Stasis' blob support is less transparent than its support for custom page layouts.

7.5 Lists

Having defined operations over `records`, we can now begin to build up more complex data structures. Rather than borrow data modeling approaches from the database literature, Stasis' default data structures borrow heavily from programming languages' collection APIs.

Linked lists are perhaps the easiest data structure to support; they only require `record` allocation, reads, and updates. The simplest approach simply calls `Talloc()` each time a new item is to be inserted into the list, then uses `memcpy()` to combine the item and `recordids` that point to the previous and next nodes. Finally, `recordids` in the two adjacent nodes are updated using `Tset()`, and the initial value of the new item is written.

As conceptually simple as this process is, we need to introduce a new primitive in order to ensure the list performs well. `Talloc()` attempts to cluster `records` allocated by the same transaction; it has been shown that such allocation policies perform quite well, especially in the absence of additional information (15). However, in the case of our linked list, we know that adjacent records will be frequently accessed at the same time, and therefore should be collocated on disk.

The record allocator provides a method, `TallocFromPage()`, that is designed to cope with exactly this situation. It takes a pageid as well as a `record` length. If the page has enough room to store the `record`, then it allocates space and returns a new `recordid`. Otherwise, it returns `INVALID_RECORD`. Thus, the linked list implementation can arrange for adjacent records to

[2]The page allocator is currently unimplemented; its API is implemented by calling the region allocator directly.

be collocated on the same page, which improves disk locality and enables a number of lower-level optimizations (Section 7.11).

7.6 Arrays

The next data structure of interest is a growable array of fixed-length `records`. Like any other array, such structures provide O(1) lookups and updates, and are able to support efficient scans.

The crux of the problem lies in storing the offsets of each chunk of the array; one could imagine storing page-sized chunks of the array, then using a single lookup page to store the location of each chunk. Unfortunately, this disk layout degrades to that of a B-tree; eventually, the lookup page will fill up, and a second level of indirection would be required.

This problem can be avoided by allocating many contiguous pages at a time, and only storing the location of each such range (and its length) in the lookup page. Simple allocation strategies, such as doubling the size of each successively allocated range, ensure good asymptotic performance while guaranteeing that the lookup page will never fill up. Stasis' page allocator provides a method that allocates contiguous ranges of pages of arbitrary length. Stasis' array type is backed by such ranges.

7.7 Hashtables

Given the fact that Stasis provides linked lists and arrays, its transactional hashtable is straightforward. The bucket list is stored in an array, and each bucket is stored in a list. Since it is possible to efficiently layer a hashtable atop existing transactional structures, Stasis' hashtable is, in some sense, a higher-level data structure than the others in this chapter. In turn, it required no new Stasis primitives.

The hashtable is organized using a linear hash function (81). Such functions take the least N significant bits of the hashcode, and use them for the bucket address. This makes it easy to reorganize the table when it is time to add new buckets; the keys in the bucket at offset B will either end up in B, or $B + 2^N$ after the table is reorganized. Also, rather than blocking requests to the hashtable while the entire index is reorganized, linear hashes allow the table to be grown a single bucket at a time. At any given time, all buckets less than some offset B' according to the old hash function are now hashed according to the new function. This allows Stasis' hashtable to grow without bound while providing predictable latency to the application.

7.8 B-Trees

B-Trees are perhaps the most demanding of the update-in-place data structures and require a number of new Stasis primitives. For instance, as the size of a B-Tree increases, the number of buffer manager calls each index probe must make increases. This multiplies the effect of buffer manager inefficiencies and concurrency bottlenecks.

Laying data out within pages is more complex than with other data structures; since records are stored in sorted order, efficient in-memory operations that shift the slotid of ranges of slots in a page are required.

Finally, supporting efficient range queries is non-trivial; as pages are added to the B-Tree, they should be laid out in a contiguous key order; over time, node splits make this increasingly difficult. Therefore, most relational databases include a B-Tree "defrag" utility that rearranges tree indexes for reasonable scan performance.

Flash seems to solve the scan problem, but greatly complicates the problem of updating B-Tree nodes, since small, random flash writes are disproportionately expensive compared to other media, and large writes reduce the fraction of changed bytes written per I/O operation.

For these reasons, Stasis' B-Tree is still a work in progress. However, solutions to these problems are largely well-understood; the bulk of the remaining work consists of engineering tasks.

7.9 LSM-Trees

Stasis includes support for *log structured merge trees*, which translate sets of random write requests to large sequential writes. Although they give up fine grained durability, writes become available immediately (this would not be the case with a B-Tree bulk loader that was fed by a sort buffer, for instance). This largely avoids a number of problems associated with B-Trees. Scan performance is guaranteed to be optimal (since pages are laid out sequentially, and without "air holes" or other gaps. Write throughput is typically much better than that of a B-Tree, both on hard disks and current flash devices. The primary disadvantage (other than the lack of fine-grained transactions) is that random LSM-Tree lookups can be a few times more expensive than B-Tree lookups. In a well-implemented system, lookups should not be significantly more than twice as expensive.

Stasis includes two LSM-Tree implementations. The first, *Rose* (Replication-Oriented Storage Engine) is a research prototype. Its disk layout is a hybrid of column- and row-oriented approaches. It also includes superscalar compression algorithms capable of keeping up with drives in current systems, and is able to support snapshot isolation, time travel and other versioning primitives for free.

Although its implementation is well-optimized, it is still fairly immature and difficult to modify. Worse, it only supports tuples that consist of fixed-width data (though this could be changed), and table schemas must be set at compile time (this would be more difficult to change).

The second LSM-Tree implementation is designed for production use, and is simpler than Rose. In particular, it avoids hard-coding support for compression. Compared to B-Trees, LSM-Tree implementations are reasonably straightforward. However, to date, they have never been carefully optimized for a number of unnecessary constant-factor performance overheads. Also, a number of LSM-Tree primitives, such as admission control, have never been implemented. We plan to add such mechanisms to the new LSM-Tree implementation, but a number of research questions remain in this area. Chapter 10 describes Rose and LSM-Trees in more detail.

From the point of view of this discussion, it is important to note which Stasis primitives the LSM-Trees use. Rose's compressed pages store extra information (write buffers, function callbacks, and so on) for each in-memory page. It maintains this information by registering callback functions with the buffer manager.

Also, the data structures mentioned so far are update-in-place. Therefore, each time Stasis updates one of these structures, it generates a set of corresponding redo/undo log entries. LSM-Trees are log-structured; they only write to unallocated space, then atomically (by using a transaction to update a normal Stasis `record`) set pointers to the newly written data. Before the pointers can be set, the pages (which come from Stasis' page allocator) must be force written to disk; Stasis' buffer manager and allocator provide the necessary mechanisms. This allows LSM-Trees to almost entirely avoid logging while providing a limited form of durability and atomicity. Batches of updates are periodically written to disk; after a crash, the updates written will be some prefix of the updates performed at runtime.

7.10 Concurrent transactions

Stasis' data structure implementations (including the allocators) support concurrent transactions. The details of the implementation are covered in Section 9.3. For now, it is sufficient to say that data structure operations may be interleaved arbitrarily, with one important limitation: two transactions may not modify the same item in a given collection.

To see why not, consider two concurrent transactions, one of which inserts an item just before the other looks up the same item and changes its value. If the first transaction aborts, it is unclear what state the collection should be in. Worse, such schedules can lead to subtle bugs during recovery and rollback. When the first transaction rolls back, it will delete the newly created item. At this point, the second transaction's update may be lost.

Worse, if the second transaction rolls back, then it will attempt to lookup the deleted item and modify its contents. This operation is likely to fail (crash and break recovery), or cause data corruption. In fact, the same problem applies to transactions modifying `records`; in general, Stasis exhibits undefined behavior in the face of write-write conflicts.

A number of strategies allow higher-level code to handle these issues. One approach limits each transaction to a single update, and ensures that conflicting transactions never overlap. One way to ensure this is to partition the requests according to some key and then apply updates from each partition in order.

A simpler, trivially correct approach is to lock the entire database for the length of the transaction, then release the lock after the commit entry is generated, but before it reaches disk. Because the lock is not held while the synchronous disk write is being performed, this allows transactions in other threads to execute many transactions and commit them all with the next synchronous disk write. Also, since transactions are not allowed to run in parallel, such schemes do not require nested top actions or logical undo.

However, schemes based upon a single lock have a few major drawbacks. First, they do not take advantage of multicore hardware, since only one thread is active at a time. Second, if a transaction needs to read a value from disk to proceed, it blocks the entire system while waiting for the disk operation to complete.

Finally, because the lock is removed before the data reaches disk, it is possible that some other transaction will read the data before it commits. As long as such transactions do not communicate with other processes or machines, this is safe; the committing transaction is guaranteed to commit before actions taken by later transactions are made durable. However, a transaction that reads non-durable data could send it to the user or another machine, then crash before the data becomes durable. For some applications, such behavior is problematic.

The final set of approaches track the updates performed by a given request, and either avoid conflicts, or use deadlock detection to rollback conflicting transactions. Such approaches provide more opportunities for concurrency at the cost of implementation complexity, and runtime overhead.

7.11 Iterators

This chapter provided a description of data structures analogous to most commonly used collections. The remaining piece of the picture is support for iteration. Of course, iterators are needed by many applications; otherwise, it would be extremely difficult to retrieve unknown values from a hash, or to manipulate a linked list in any meaningful way. In Stasis, they perform a second, important service. Accessing Stasis' buffer manager is inherently expensive, as it involves locating a page in memory, and setting various latches.

Stasis iterator interface is designed to allow data structure implementations to safely hold page pins across invocations to `next()`. For structures that have good locality (as all of the ones mentioned in this chapter do), this amortizes away most of Stasis' overhead. Recall that set-oriented processing (Section 4.3.5) has been a crucial performance optimization throughout the history of databases.

Since application code is running in the same process as Stasis, there is no need to ship computation, as many database systems do. Iterators are a natural way to deal with collections of data and are the default approach; systems based upon event-driven callbacks would be more akin to traditional server-side database processing and are easily supported.

In addition to read-only iterators, it is possible to implement iterators that allow data to be manipulated as they scan data structures. However, just as with in-memory iterators, this can lead to subtle problems if manipulating data changes its position in the iterator.

A final related primitive is used by Rose, which makes use of trees that provide append operations that can compress a tuple and append it to the end of a tree faster than Stasis' buffer manager can pin a page. These systems provide "append" operations akin to an iterator. Unlike a normal insertion method, callers of append must obtain a "write handle" before inserting tuples, then close the handle once the last tuple is inserted.

Write handles typically contain a page offset (and pin), as well as the slot number of the last value on the page. Caching this state reduces the cost of inserting each record. Closing the write handle frees memory and releases any page pins still held by the handle.

7.12 Two-phase commit and other protocols

It often makes sense to compose Stasis with other storage systems, such as file systems, databases, or simply other instances of Stasis. Each such system speaks a separate, and often application-specific, network protocol. A *transaction coordinator* is a service that atomically commits transactions that run across multiple storage systems. Protocols such as XA (57) standardize transaction coordinators.

Although Stasis is not compliant with the XA specification, it is designed to act as both a transaction coordinator and as a participant. This is implemented using two mechanisms. The first is a special type of log entry that *prepares* a transaction to commit. This is equivalent to

	Index				
Stasis primitive	Arrays	Lists	Hashtables	B-Trees	LSM-Trees
Introspection	Y				Y
Record allocation		Y	Y		
Page allocation	Y	Y	Y	Y	
Region allocation	Y		Y		Y
Allocate from page		Y	Y	Y	
Record reordering within pages				Y	
Page allocator locality hints		Y	Y	Y	
Buffer manager callbacks					Y
Log structured updates					Y
Iterators	Y	Y	Y	Y	Y

Table 7.1. Primitives used by Stasis' index implementations

two-phase commit's prepare entry. Once a transaction has been prepared, if the system crashes, the transaction will be revived and be able to continue once Stasis recovery has completed.

In of itself, this is not particularly useful, as prepared transactions often have some state (such as a list of participants, or information for non-transactional systems) associated with them. Therefore, Stasis' transaction table allows prepared transactions to associate a bit of state with their transaction table entry. Transactions must log this state themselves; having done so, they are guaranteed that it will be available after a crash.

Consider a Stasis transaction that is coordinating between a database and a file server. It would first send some data to the file server and some queries to the database. The files would be written to a temporary location. Next, it would initiate a round of two phase commit, and instruct the file server to force write the files, and ask the database to prepare to commit the transaction. Upon success, it would tell the file server to rename the files and tell the database to commit the transaction. If the coordinator crashed during this process, recovery would revive the transaction, and the coordinator would discover the file names and remote transaction ID in the transaction table.

7.13 Summary

The data structures described in this chapter provide an overview of the storage systems, open source and proprietary, built upon Stasis to date. Table 7.1 provides a summary of the mechanisms used by each index.

This chapter provided an overview of the high-level primitives exposed by Stasis. The APIs described here are still evolving, and Stasis' online documentation remains the canonical reference. However, the types of primitives exposed by Stasis are likely to remain relatively stable over time; this chapter is likely to be an appropriate starting place for system designers for the foreseeable future. The following chapters turn to a more detailed discussion of Stasis' lower-level APIs, beginning with an explanation of the programming model provided for programmers building new concurrent transactional data structures on top of Stasis.

Chapter 8

Update-in-place examples

Stasis provides applications with the ability to customize storage routines and recovery semantics. This chapter describes a number of such extensions, and shows that this flexibility does not come at a significantly higher cost than existing general-purpose transactional storage implementations. In order to illustrate the advantages of using Stasis' lower-level interfaces, this chapter also illustrates a number of optimizations that rely upon techniques that will be covered in later chapters.

8.1 Experimental setup

We chose Berkeley DB in the following experiments because it provides transactional storage primitives similar to Stasis, is commercially maintained and is designed for high performance and high concurrency. For all tests, the two libraries provide the same transactional semantics unless explicitly noted.

The experiments in this chapter were run on an Intel Xeon 2.8 GHz processor with 1GB of RAM and a 10K RPM SCSI drive using ReiserFS (111).[1] All results correspond to the mean of multiple runs with a 95% confidence interval with a half-width of 5%.

Our experiments use Berkeley DB 4.2.52 with the flags DB_TXN_SYNC (force log to disk on commit), and DB_THREAD (thread safety) enabled. We increased Berkeley DB's buffer cache and log buffer sizes to match Stasis' default sizes. If Berkeley DB implements a feature that Stasis is missing we enable it if it improves performance.

We disable Berkeley DB's lock manager for the benchmarks, though we use "Free Threaded" handles for all tests. This significantly increases performance by eliminating transaction deadlock, abort, and repetition. However, disabling the lock manager caused concurrent Berkeley DB benchmarks to become unstable, suggesting either a bug or misuse of the feature. With the lock manager enabled, Berkeley DB's performance in the multi-threaded benchmark (Section 8.2) strictly decreased with increased concurrency.

We expended a considerable effort tuning Berkeley DB and our efforts significantly improved

[1]We found that the relative performance of Berkeley DB and Stasis under single-threaded testing is sensitive to file system choice, and we plan to investigate the reasons why the performance of Stasis under ext3 is degraded. However, the results relating to the Stasis optimizations are consistent across file systems.

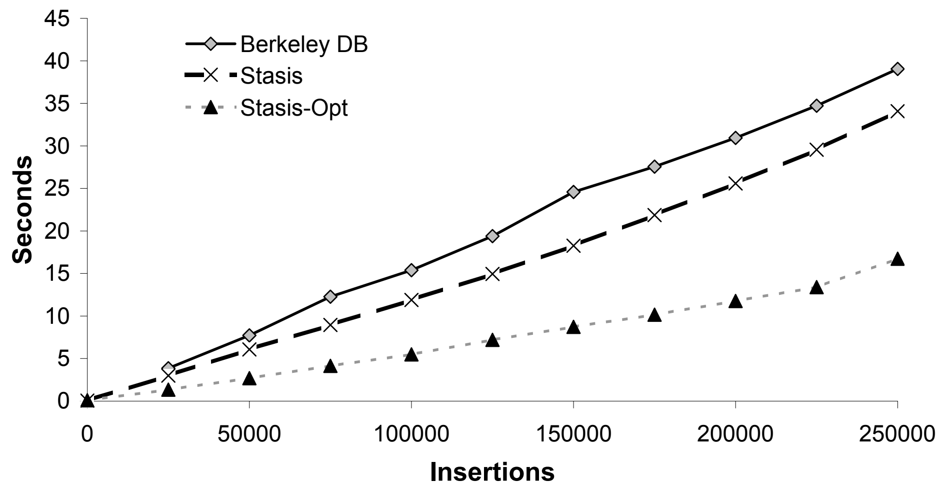


Figure 8.1. Performance of Stasis and Berkeley DB hash table implementations. The test is run as a single transaction, minimizing synchronous log writes.

Berkeley DB's performance on these tests. Although further tuning by Berkeley DB experts would probably improve Berkeley DB's numbers, we think our comparison shows that the systems' performance is comparable. As we add functionality, optimizations, and rewrite modules, Stasis' relative performance varies. We expect Stasis' extensions and custom recovery mechanisms to continue to perform similarly to comparable monolithic implementations.

8.2 Linear hash table

This section presents two hash table implementations built on top of Stasis, and compares them with the hash table provided by Berkeley DB. One of the Stasis implementations is simple and modular, while the other is monolithic and hand-tuned. Our experiments show that Stasis' performance is competitive, both with single-threaded and high-concurrency transactions.

The modular hash table uses nested top actions to update its internal structure atomically. It uses a linear hash function, allowing it to increase capacity incrementally, and is based on a number of sub-modules, as described in Section 7.7.

The hand-tuned hash table is also built on Stasis and also uses a linear hash function. However, it is a monolithic mockup that used carefully ordered writes to reduce runtime overheads such as log bandwidth. Berkeley DB's hash table is a popular, commonly deployed implementation, and serves as a baseline for our experiments.

Both of our hash tables outperform Berkeley DB on a workload that populates the tables by repeatedly inserting $(key, value)$ pairs (Figure 8.1). The performance of the modular hash table shows that data structure implementations composed from simpler structures can perform comparably to the implementations included in existing monolithic systems. The hand-tuned implementation shows that Stasis allows application developers to optimize important primitives. However, it resorts to optimizations that are neither easily tested nor general purpose. Section 14.2 describes a number of low-level, general purpose techniques that can be used to reduce the performance gap between the modular implementation and the hand tuned one.

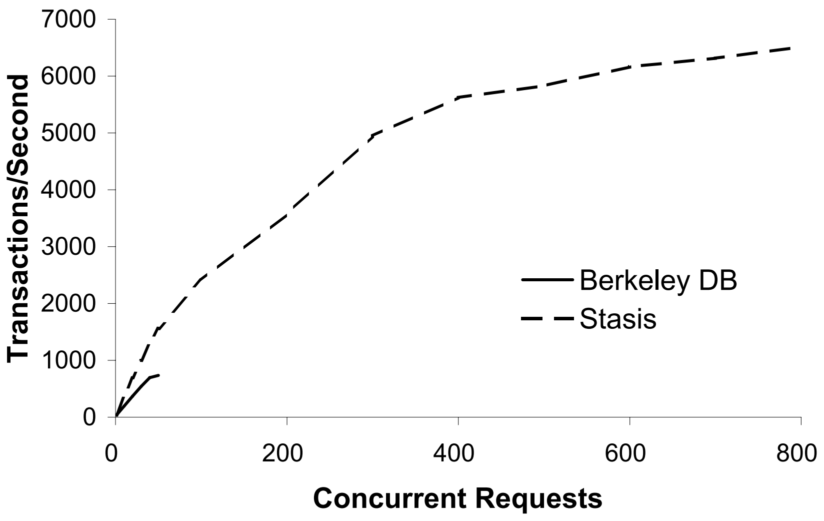


Figure 8.2. High-concurrency hash table performance. Our Berkeley DB test can only support 50 threads (see text).

Figure 8.2 describes the performance of the two systems under highly concurrent workloads using the ext3 file system (the experiments predate ext4).[2] This test uses the modular hash table to demonstrate the performance of a simple, clean data structure implementation that a typical system implementer might produce, not the performance of our own highly tuned implementation.

Both Berkeley DB and Stasis can service concurrent calls to commit with a single synchronous I/O. Stasis scaled quite well, delivering over 6000 transactions per second, and provided roughly double Berkeley DB's throughput (up to 50 threads). Although not shown here, we found that the latencies of Berkeley DB and Stasis were similar.

8.3 Object persistence

Two different styles of object persistence have been implemented on top of Stasis. The first object persistence mechanism, pobj, provides transactional updates to objects in Titanium, a Java variant. It transparently loads and persists entire graphs of objects, but will not be discussed in further detail. The second variant was built on top of a C++ object persistence library, Oasys. Oasys uses plug-in storage modules that implement persistent storage, and includes plugins for Berkeley DB and MySQL. Like C++ objects, Oasys objects are explicitly freed. However, Stasis could also support concurrent and incremental atomic garbage collection (71).

This section describes how the Stasis plugin supports optimizations that reduce the amount of data written to log and halve the amount of RAM required. We present three variants of the Stasis plugin. The basic one treats Stasis like Berkeley DB. The "update/flush" variant makes use of Stasis *segment-based recovery* algorithm which is described in more detail in Chapter 11. Finally, the "delta" variant uses update/flush, but only logs the differences between versions.

The update/flush variant allows the buffer manager's view of live application objects to become

[2]Multi-threaded benchmarks were performed using an ext3 file system. Concurrency caused both Berkeley DB and Stasis to behave unpredictably under ReiserFS. Stasis' multi-threaded throughput was significantly better than Berkeley DB's with both file systems.

stale. This is safe since the system is always able to reconstruct the appropriate page entry from the live copy of the object. This reduces the number of times the plugin must update serialized objects in the buffer manager, and allows us to nearly eliminate the memory used by the buffer manager.

We implemented the Stasis buffer pool optimization by adding two new operations, `update()`, which updates the log when objects are modified, and `flush()`, which updates the page when an object is evicted from the application's cache.

The third plugin variant, "delta," incorporates the update/flush optimizations, but only writes changed portions of objects to the log. With Stasis' support for custom log formats, this optimization is straightforward.

Oasys does not provide a transactional interface. Instead, it is designed to be used in systems that stream objects over an unreliable network connection. The objects are independent of each other, so each update should be applied atomically. Therefore, there is never any reason to roll back an applied object update. Furthermore, Oasys provides a sync method, which guarantees the durability of updates after it returns. In order to match these semantics as closely as possible, Stasis' update/flush and delta optimizations do not write any undo information to the log. The Oasys sync method is implemented by committing the current Stasis transaction, and beginning a new one.

As far as we can tell, MySQL and Berkeley DB do not support this optimization in a straightforward fashion. "Auto-commit" comes close, but does not quite provide the same durability semantics as Oasys' explicit syncs.

The operations required for the update/flush and delta optimizations required 150 lines of C code, including whitespace, comments and boilerplate function registrations. Although the reasoning required to ensure the correctness of this optimization is complex, the simplicity of the implementation is encouraging.

In this experiment, Berkeley DB was configured as described above. We ran MySQL using InnoDB for the table engine. For this benchmark, it is the fastest MySQL engine that provides similar durability to Stasis. We linked the benchmark's executable to the `libmysqld` daemon library, bypassing the IPC layer. Experiments that used IPC were orders of magnitude slower.

Figure 8.3 presents the performance of the three Stasis variants, and the Oasys plugins implemented on top of other systems. In this test, none of the systems were memory bound. As we can see, Stasis performs better than the baseline systems, which is not surprising, since it exploits the weaker durability requirements.

In non-memory bound systems, the optimizations nearly double Stasis' performance by reducing the CPU overhead of marshaling and unmarshaling objects, and by reducing the size of log entries written to disk.

To determine the effect of the optimization in memory bound systems, we decreased Stasis' page cache size, and used O_DIRECT to bypass the operating system's disk cache. We partitioned the set of objects so that 10% fit in a *hot set*. Figure 8.3 also presents Stasis' performance as we varied the percentage of object updates that manipulate the hot set. In the memory bound test, we see that update/flush indeed improves memory utilization.

As above, Stasis outperformed existing systems before we applied sophisticated optimizations. While an understanding of the material in the following chapters will allow developers to signifi-

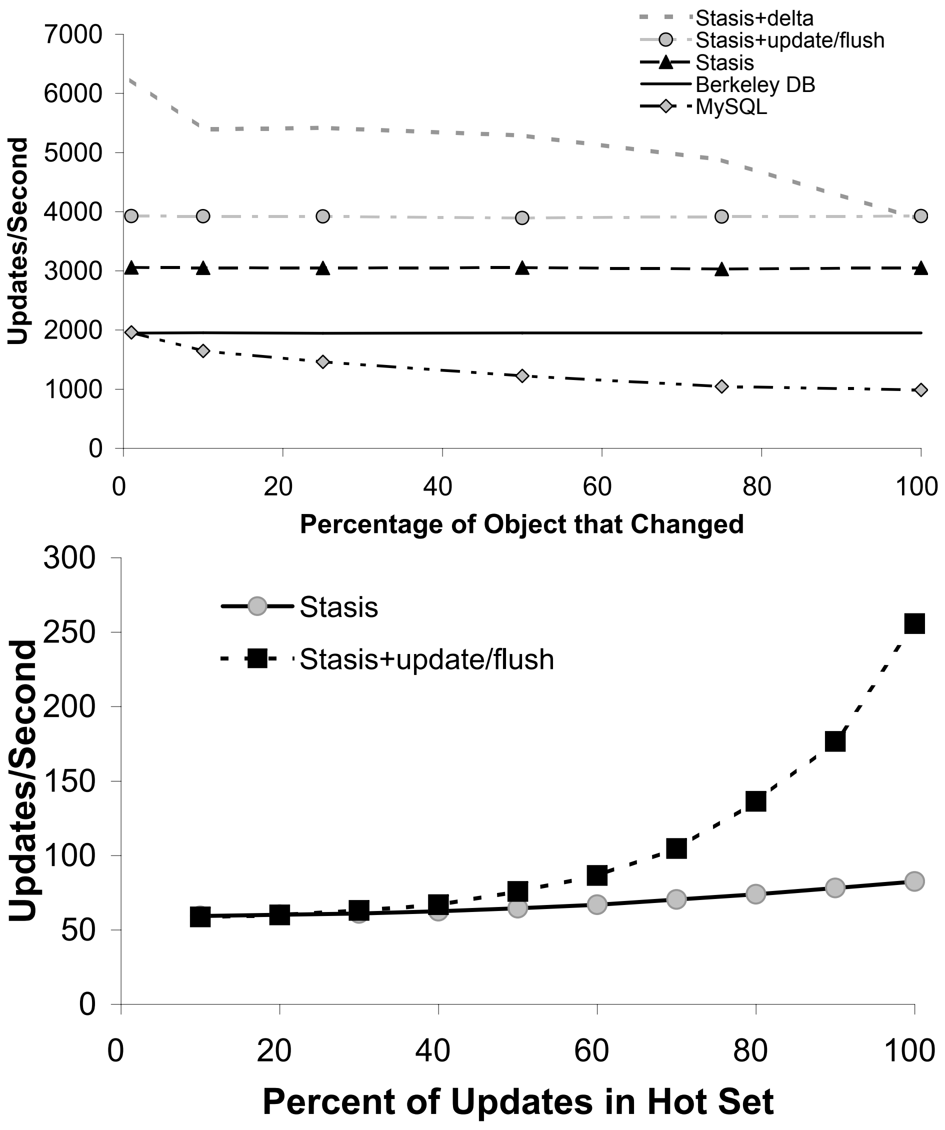


Figure 8.3. The effect of Stasis object-persistence optimizations under low and high memory pressure.

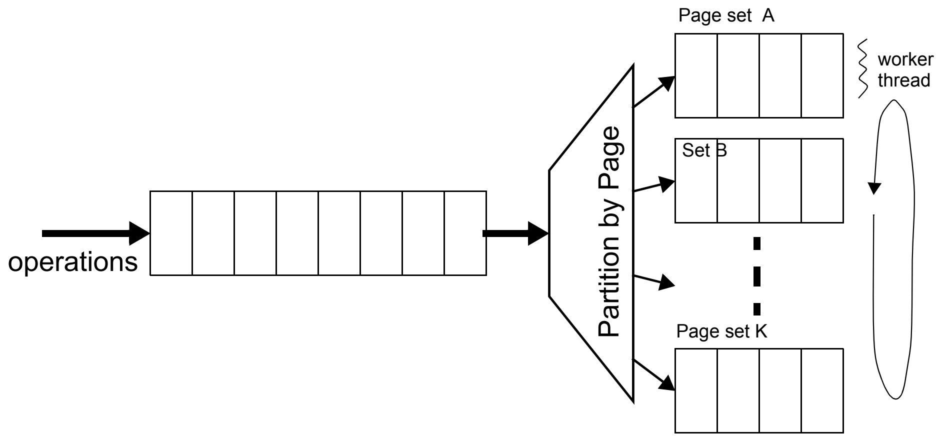


Figure 8.4. Locality-based request reordering. Requests are partitioned into queues. Queue are handled independently, improving locality and allowing requests to be merged.

cantly improve application performance, one can forgo such optimizations and still obtain performance that is competitive with existing approaches.

8.4 Request reordering

In this experiment, we are interested in allowing Stasis to efficiently manipulate sequences of application requests. By translating these requests into logical operations (such as those used for logical undo), we can manipulate and optimize such requests. Because logical operations generally correspond to application-level operations, application developers can easily determine whether logical operations may be reordered, transformed, or even dropped from the stream of requests that Stasis is processing. For example, requests that manipulate disjoint sets of data can be split across many nodes, providing load balancing. Requests that update the same piece of information can be merged into a single request; RVM's "log merging" implements this type of optimization (114). Stream aggregation techniques and relational algebra operators could be used to transform data efficiently while it is laid out sequentially in non-transactional memory.

To experiment with the potential of such optimizations, we implemented a request reordering scheme that increases request locality during a graph traversal. The graph traversal produces a sequence of read requests that are partitioned according to their physical location in the page file. Partition sizes are chosen to fit inside the buffer pool. Each partition is processed until there are no more outstanding requests to read from it. The process iterates until the traversal is complete.

We ran two experiments. Both stored a graph of fixed-size objects in the growable array implementation that is used as our linear hash table's bucket list. The first experiment (Figure 8.5) is loosely based on the OO7 database benchmark (25). We hard-code the out-degree of each node and use a directed graph. Like OO7, we construct graphs by first connecting nodes together into a ring. We then randomly add edges until obtaining the desired out-degree. This structure ensures graph connectivity. Nodes are laid out in ring order on disk so at least one edge from each node is local.

The second experiment measures the effect of graph locality (Figure 8.6). Each node has a distinct hot set that includes the 10% of the nodes that are closest to it in ring order. The

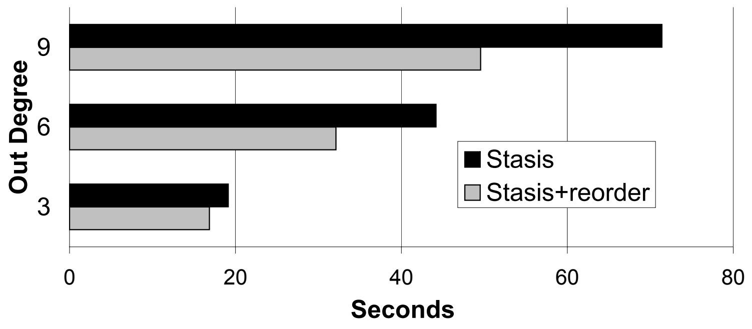


Figure 8.5. OO7 benchmark style graph traversal. The optimization performs well due to the presence of non-local nodes.

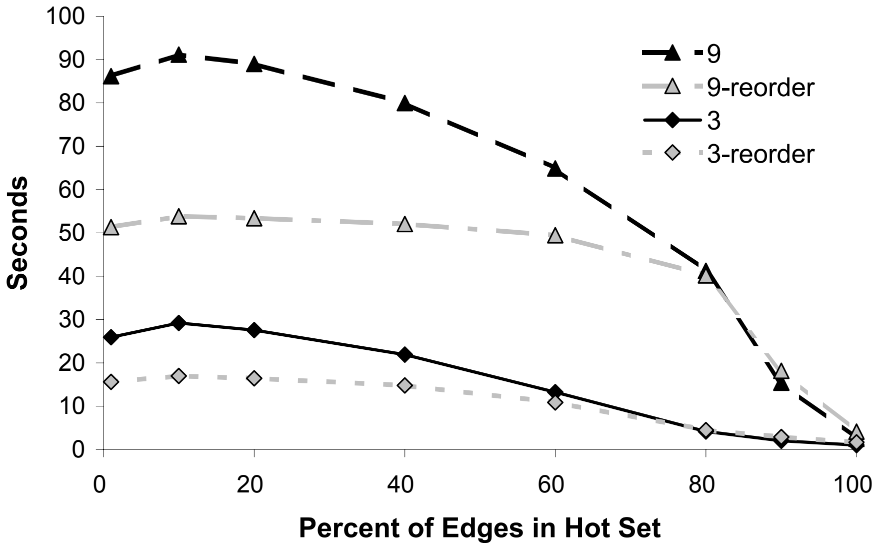


Figure 8.6. Hot-set based graph traversal for random graphs with out-degrees of 3 and 9. The multiplexer has low overhead, and improves performance when the graph has poor locality.

remaining nodes are in the cold set. We do not use ring edges for this test, so the graphs might not be connected. We use the same graphs for both systems.

When the graph has good locality, a normal depth-first search traversal and the prioritized traversal both perform well. As locality decreases, the partitioned traversal algorithm outperforms the naive traversal.

Note that these optimizations have minimal interaction with Stasis' recovery algorithms; instead, they reuse Stasis logging primitives in an application-specific way. Section 11.7 describes a number of experiments that use segment-based recovery to reorder and coalesce requests. Such techniques require an in-depth understanding of Stasis' recovery algorithms, but provide more flexibility than the techniques employed here.

Chapter 9

Extending Stasis with custom operations

9.1 The Stasis programming model

This chapter describes Stasis *operations*, the building blocks from which indexes and other concurrent transactional data structures are built. A Stasis operation consists of a redo and undo method and a corresponding log entry format. In order to make effective use of Stasis operations, one must first understand how to write multithreaded software; Stasis' undo methods are analogous to thread safe exception handlers in modern languages. Before discussing concepts specific to Stasis, transactions or storage, this section turns to a brief introduction to multithreaded programming.

9.2 Consistency: Latching to maintain invariants

In ACID transactions, "C" stands for consistency, the idea that each transaction takes the system from one consistent state to another. The exact semantics of "consistent" are left up to the data model and the application. In concurrent programs, latches are typically used for the same purpose. In fact, it is difficult to imagine a useful application of latching that falls outside this broad definition.

Latch ordering is a latching protocol that covers the vast majority of concurrent programming scenarios. The idea is simple: one first defines a partial ordering (ie: a lattice) over all of the latches in the system. The latches held by a given thread at a given time correspond to some path along the lattice. One must obtain the latches in order; the thread walks the lattice and obtains the latch corresponding to each node as it is encountered.

When the thread decides to release a latch, it has two choices. With the first approach, each thread must release some suffix of the path before obtaining additional latches. They may then safely obtain additional latches by tracing a path starting at the latch that is still held.

With this first approach, the "path" taken through the lattice is actually a stack. This has two convenient consequences. First, there is a natural correspondence between latch orderings and threads' runtime stacks; rather than represent paths explicitly in some auxiliary data structure, one can simply ensure that each stack frame releases each latch it acquired.

Also, standard programming approaches based upon encapsulation lead to a powerful analysis tool for concurrent programs. Assuming that each object (or module) has a corresponding latch, and that each call that obtains a latch releases it before returning, it is easy to locate potential deadlocks in the system. One need only check for "upcalls" where a module invokes a method from some other module that in turn calls back into the first module. Assuming there are no such calls, the program is deadlock free. As we will see below, this approach is the one used by Stasis' multilevel recovery scheme.

The second approach to releasing latches treats the path through the lattice as a queue; latches are released in the same order they were obtained. This approach is called "lock crabbing." It is particularly useful when implementing concurrent data structures; hashes and linked lists are typical examples.

Note that while these protocols are similar to *two-phase locking*, which is commonly used to protect resources in databases, they differ in a few important ways. Two-phase locking also defines lock orders and manages the locks as a stack. However, it is designed to avoid the cost of obtaining many fine-grained latches when manipulating a significant fraction of the database without sacrificing concurrency for fine-grained operations (52).

9.2.1 Spinning: Transactions in disguise

Often, it is difficult to establish a latch order, or it is important that latches are not held during expensive operations. Unless carefully handled, such situations lead to deadlocks or races.

Unfortunately, while both problems are automatically handled by transactional concurrency control, such mechanisms are significantly more expensive than modern thread synchronization primitives, and are of little use unless read and write requests can be automatically intercepted and checked. At any rate, building Stasis atop another transaction system would simply push the problems down into another system; another approach is needed.

Transactional concurrency control mechanisms deal with deadlocks and recover from races by repeatedly rolling back and retrying requests. This trick works for multithreaded programming as well; it is common to acquire a mutex, then check that some invariant holds. As long as the invariant does not hold, the system repeatedly releases and reacquires the mutex, perhaps by waiting on a condition variable, or by performing some expensive operation, such as a disk I/O when the latch has been released.

Code that must resort to this approach is rare; only a few Stasis modules spin on latches in this way. In circumstances when a latch should not be held across an expensive operation, it is often preferable to use finer-grained latches. After all, as long as the threads that block on the finer-grained latch must also block on the expensive operation, there is no reason to avoid holding the latch while the operation completes. In the common case, acquiring a fine-grained latch is still inexpensive, and allows for greater concurrency than the coarse-grained approach.

9.3 Logical undo and concurrent transactions

The previous section described a range of potential latching mechanisms. Stasis' data structure implementations make use of a simple approach called *multilevel recovery* (141): each time a transaction is done manipulating a data structure, it releases the latch protecting the structure. Just before the latch is released, the transaction appends a logical undo to the log. Latches are obtained in some order and managed as a stack; the most recently acquired latch is the first to be released.

This section explains how the protocol works, and why logical undo is necessary in the first place. Just as with multithreaded programs, transactional data structures use latches to protect internal invariants. While a latch is being held, various redo and undo entries are generated, and data structure invariants are temporarily broken. When the latch is released, the data structure is consistent, and other transactions may modify the data structure, including `records` modified by the first transaction.

Consider a transaction that modifies a B-Tree and logs physical redo and undo entries that correspond to the changes made to tree nodes. It is possible that another transaction will rebalance the tree nodes before the first transaction completes. If so, the log entries generated by the first transaction will no longer point to appropriate nodes. If the first transaction rolls back at this point, it will apply the entries and corrupt the tree.

This is where the need for logical undo comes in; rather than remember exactly which nodes were modified by the first transaction, the log should contain a description of the keys and values that were inserted into the tree.

By generating the entries before releasing the latch, the tree implementation ensures that concurrent transactions do not modify the tree and invalidate the physical undo entries until the logical undo is in place.

The underlying log mechanism that supports such operations is called a *nested top action*; each time a transaction begins a data structure operation that should be protected by a latch, it first obtains the latch, then begins a nested top action. Just before releasing the latch, it writes a logical undo, ending the nested top action.

Note that nested top actions may be nested; in fact, operations within a transaction can form arbitrary tree structures. However, nested top actions incur significant log overheads; if the outer operation is protected by a single latch, then the inner nested top actions serve no useful purpose, and can be omitted.

With care, finer-grained data structure latching may be performed as well, just as with multithreaded programs. However, transactional structures introduce extra complexity; cases that involve concurrent transaction rollbacks must be considered as well. Rather than work through the corner cases from scratch, it is probably better to consult one of the well-known index latching protocols (79; 72).

Indexes based upon multilevel recovery are easily implemented and allow transactions that manipulate the same indexes to run in parallel. However, they rely upon coarse-grained index latches (but *not* transaction duration locks), which prevents multiple threads from manipulating the same structure at once. More complex latching protocols, such as those used for B-Trees, are both well-understood and complex. Taken together, these approaches support a wide-range of concurrent workloads without the need for unduly complex application logic.

9.4 The need for lower-level APIs

Higher-level code typically calls Stasis methods that automatically pin pages, generate log entries, then unpin the pages. Such methods' names are prefixed with a `T`; the ones that manipulate a record at a time are named `Tset()` and `Tread()`. These high-level methods are meant to be comparable to the API's exposed by systems such as BerkeleyDB, albeit without lock management, or other concurrency control mechanisms.

Stasis provides a full range of primitives for higher-level code; in fact, it is possible to implement many indexes atop such structures without making direct use of the buffer manager API. In order to do so, one need only implement data structure methods using high-level interfaces, then protect methods that should be atomic with latches and nested top actions. In fact, some of Stasis' data structures are implemented this way.

However, layering atop high-level APIs has a number of disadvantages; the most important are:

Limited concurrency Most of Stasis' general purpose calls are not commutative. If an application decides to increment some `record` using `Tread()` and `Tset()`, it must protect the `record` with a transaction-duration lock, or risk having some other transaction modify the value, and break rollback and recovery.[1] Nested top actions provide one solution, but make use of more log bandwidth than is strictly necessary. Logging custom physical redos and undos saves significant log space when only a single page is modified. If multiple pages are to be touched by the same commutative operation, then a nested top action is probably necessary.

Excessively large log entries `Tset()` has significant overhead; it writes the pre- and post-value of the record it modifies. In some circumstances, such as updating a field in a record, less information could be logged. Since Stasis treats `records` as opaque data, in order for an application to perform such optimizations, it must first "teach" Stasis about the data layout formats using custom operations. The alternative, hard-coding Stasis to a particular data model, would lead to impedance mismatch and compromise Stasis' flexibility.

Too many log entries Each `Tset()` generates a new log entry. If many `Tset()` operations are performed against a single page by a single transaction then they can be coalesced into a single entry. This is particularly important for index reorganization and other bulk operations that have good physical locality.

Lack of expressiveness Stasis' page API (Section 9.7) provides a number of page layouts that address the needs of a wide range of indexes and other data structures. The page API also allows new page types to be added, allowing it to support indexes that require more control over disk layouts. However, situations where the page API is of little use or is inadequate inevitably arise. Bypassing or extending the page API typically addresses such situations.

Taken together, these limitations often form a compelling set of reasons to deal with the complexity of manual buffer management and log entry generation. Conceptually, working with the lower-levels APIs is similar to building data structures out of higher-level data structures, except that one must deal with wider-variety of lower-level mechanisms. The following sections explain how to implement *custom operations* directly on top of Stasis' buffer manager and log.

[1]Stasis actually contains two methods `Tincrement()` and `Tdecrement()` to address this specific situation, but it is impossible to anticipate all forms of commutativity and weakened consistency that may arise in the future.

9.5 Buffer management

Before accessing a disk page that is managed by Stasis, custom operations must pin the page. This provides a C pointer to the in-memory copy of the page. The page contents consist of an (optional) Stasis-defined header, and an opaque array of bytes. Having pinned a page, the logical operation is free to manipulate the data it contains as it sees fit. The only constraint is that the changes to the page be repeatable and reversible using a pair of physical (or physiological) redo and undo entries (Section 9.6). If the operation decides to modify the page, then it must inform the buffer manager before unpinning it.

9.5.1 Page latches

While a pinned page is being modified, it is generally important that no other threads attempt to manipulate the page. Each page in the buffer manager has a read/write (shared/exclusive) latch associated with it that protects shared state, such as the page header, from concurrent manipulation.

Depending on the exact layout of the page, the latch may be unnecessary. Recall that Stasis transactions exhibit undefined behavior in the face of concurrent transactions that modify the same data item. The same is true for Stasis' buffer manager and page manipulation code. Some page layouts make use of immutable headers; since the header information can never change, there is no reason to latch it. Assuming that higher-level concurrency control mechanisms ensure that no two threads concurrently read and write the same record at the same time, then the only page state that must be latched is the LSN. Stasis' buffer manager and dirty page table protect against concurrent attempts to update the LSN. Therefore, no page latch is necessary.

9.5.2 Page latch ordering

Stasis' initial buffer manager allowed each thread to latch one page at a time. Although it simplified some aspects of the buffer manager, it proved to be incredibly inconvenient. The second (and extremely short-lived) iteration of Stasis' buffer manager allowed multiple pages to be pinned at once, but (accidentally) imposed latch and pin ordering constraints upon callers. This proved to be completely unworkable. Today, Stasis' buffer manager allows threads to pin arbitrary numbers of pages at once, up to the size of memory. In addition to simplifying code that uses lock crabbing to manipulate sets of pages, this adds support for no-Steal transactions.

Also, avoiding constraints upon page latch orders is crucial for indexes such as B-Trees, that have natural orderings that are not easily circumvented. However, it does not save operations that pin multiple pages at once from avoiding latch deadlocks. Stasis' latches are based upon mutexes (not lock management, or other transaction-oriented concurrency control schemes). Therefore, Stasis latches must be treated like any other mutex-based concurrency control mechanisms; deadlocks due to improper latching protocols will crash the system. However, all of the standard approaches to multithreaded programming still apply (Section 9.2).

9.6 Log entries

Up to this point, this chapter has treated log entries in vague terms. This section explains exactly what custom operations must do to generate log entries, then defines the three commonly used types of log entries that were alluded to above.

In order to generate a log entry, custom operations specify an opcode, which is simply a globally unique integer that refers to the operation in question. Associated with this opcode are a few pieces of additional information:

An undo opcode specifies an operation that, given the log entry, will reverse the action the operation took.

A redo opcode is generally the same as the log entry's opcode, but allows different operations to be taken at recovery and during forward operation.

Redo and undo callbacks are C function pointers that Stasis invokes when appropriate.

The custom operation must also specify the location of the page that it should be applied to, and a byte array that contains any additional information that will be needed to apply the log entry. Stasis' `Tset()` method includes the pre-image and post-image of the modified record in this byte array. The size of each log entry is recorded in the log for internal bookkeeping purposes; simple arithmetic allows `Tset()`'s operations to compute the size of the `record` the log entry refers to, then compare it against the size of the `record` in the buffer manager. Barring bugs in Stasis or in higher-level concurrency control mechanisms, the two entries are guaranteed to match; in an ideal world, the comparison would be redundant.

The three types of log entries that custom operations generate can be described in these terms. The first type, *physical redo*, specifies a single page that will be modified. Given the contents of the log record, and of the page as it existed immediately before the entry was applied, the redo's callback function makes the necessary change to the page. Redo entries also specify undo callbacks, which are analogous; the primary difference is that, given the version of the page after redo, they revert the page to its old value.

The second type, *physiological redo*, allows for two versions of the page to be "equivalent," even though they do not match bit-for-bit. This is important for page layouts such as slotted pages (Section 9.7.1) that contain headers or other mappings between slots (or other record identifiers), and the offset of the record. Since such formats typically can be defragmented, simply writing the physical offset of the underlying record to the log would be inadequate.

Physiological redo also allows for commutative operations. Stasis does not actually reorder commutative operations (except when segments are in use; see Section 11.4). Instead, commutativity comes into play because it allows Stasis to treat modifications to the record that were performed after redo, but before undo as though they happened before the redo. This allows concurrent transactions to safely write to `records` that were updated by transactions that have not yet completed.

Finally, physiological redo allows for some degree of reuse across operation implementations. When Stasis loads pages from disk, it determines the type of each page. Attempts to access the page are then routed to callbacks associated with the page type. This allows a single implementation of a custom operation to be applied to a wide variety of page layouts, and is how `Tset()` is able to support many different page types without defining multiple log entry formats.

	Page layouts			
Methods	Fixed	Slotted	Raw	Compressed
Read record (by slot)	Y	Y		Y
Append record	Y			Y
Write record (by slot)	Y	Y		
Allocate record		Y		
Free record		Y		
Iterator	Y	Y		Y
Reorder records		Y		
Find record (by value)		Y		Y
Insert sorted record		Y		

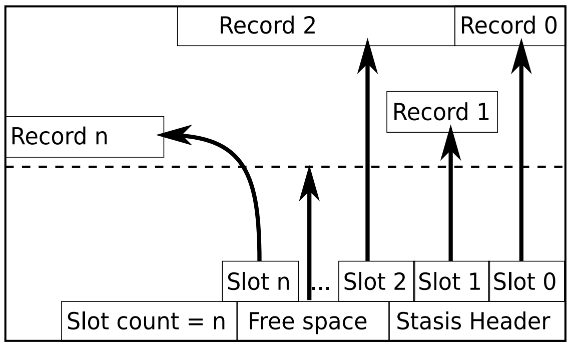
Table 9.1. A summary of the methods provided by Stasis' page layouts.

The final type, *logical undo*, does not specify a page to be manipulated. Instead, it describes the inverse of some high-level operation (such as inserting a value into a tree). Its callback uses the contents of the log entry to locate the data structure that was modified, and to undo the change. Typically, such callback implementations simply invoke the method that the application would use in order to undo its changes. Note that this method will generate log entries, even if invoked during recovery. Stasis' recovery algorithms require that logical undo operations generate such entries, regardless of whether the transaction rolls back during recovery or normal operation. In contrast, physical undos must not generate log entries, as Stasis produces such entries automatically.

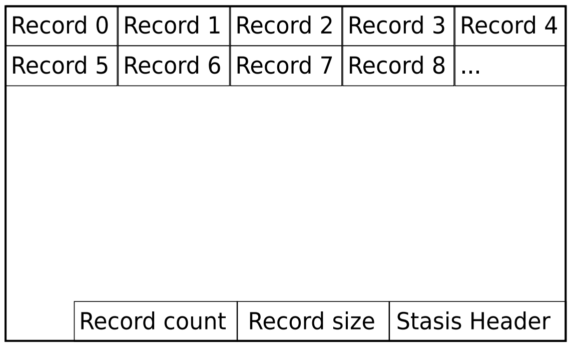
9.7 Page layouts

Stasis' page layouts define the organization of data within each page. Most layouts implement some subset of Stasis' standard page operations, which include methods such as "set `record`", "available space," "allocate `record`," and so on. Layouts implement subsets of these records because specialized page formats often lack support for certain operations. For example, Stasis' *fixed* pages store arrays of fixed-length records. They cannot provide a "deallocate" method because they do not pay the space overhead of tracking which records are currently in use. Similarly, Stasis' compressed pages are append only, and do not provide standard "set `record`" methods.

In some respect, page layouts are based upon an *algebraic* type system, rather than an object-oriented one based upon inheritance. This is consistent with data structure libraries such as C++'s STL, and most strongly typed functional languages based upon ML. However, it differs from Java's collection API, which is familiar to a wide range of programmers. The distinction between the two approaches is often a matter of taste. However, algebraic types are more easily expressed in C than inheritance hierarchies. Also, over time, methods have been added to Stasis' page layout interface. If Stasis page layouts were defined in terms of inheritance, then retrofitting in such methods would likely have required modifications to the inheritance relationships between the existing formats. Since page layouts are based upon algebraic types, these modifications were straightforward. Table 9.1 provides an overview of Stasis' predefined page formats and the methods that each supports.



(a) Slotted pages store variable length data.



(b) Fixed pages have minimal space overhead, but only support a single record size per page, and have limited support for allocation.

Figure 9.1. Stasis page layouts

9.7.1 Slotted

Stasis' slotted pages are used by default, and are the most general purpose of Stasis' predefined page layouts. They include a header that points to variable length records that are stored within the page (Figure 9.1(a)). When the free space of the page becomes fragmented it can be reorganized without writing additional log entries. Since the slots of each record remain the same after reorganization, and redo and undo entries index into pages using the slot number, reorganizing pages does not affect recovery. This is particularly important for B-Tree implementations, which must routinely insert and remove slots from the middle of a page. Such operations lead to internal page fragmentation and frequent reorganization. B-Tree's require a number of other special methods, since they read, insert and remove slots by value while ensuring that the page remains in sorted order. Such methods could be added to Stasis' page layout interfaces, but currently are layered on top of more general primitives that insert and remove ranges of pages in between existing slots.

9.7.2 Fixed

Stasis' *fixed* pages provide most of the same methods as slotted pages, with two exceptions. First, each page only holds slots of some pre-defined length. Second, instead of tracking the status of each record on the page, fixed pages simply track the highest numbered slot (Figure 9.1(b)). This reduces the size and complexity of the page header significantly. Also, if the records on the page are pre-allocated when the page is initialized, this avoids the need for page latches that protect the header from concurrent modifications.

9.7.3 Index headers

Index header pages typically reuse Stasis' slotted or fixed page layouts, with a few modifications. First, rather than store "slotted" or "fixed" as the page type, they define a new page type that describes the index header, and use that to describe the page. Second, they register buffer manager callbacks that will be automatically invoked. The first is called each time the page is loaded. It examines the page, and takes actions such as precomputing values, allocating scratch space, copying statistics into in-memory structures and so on. The second is called each time the page is written back to disk. This call updates the page, if necessary, usually by copying data from scratch space into the page. The third call is invoked when the page is evicted from memory. It frees any memory and releases any resources that are associated with the page.

Is important to note that producing custom page formats for index headers is entirely optional; many of Stasis' indexes avoid doing so. However, in some circumstances, custom formats can significantly reduce runtime overheads and simplify management of various background tasks.

Finally, custom formats with no purpose other than to modify the page type are often useful for introspection and for utilities such as `fsck`. They are used by Stasis' array implementation as follows: each time a slot is requested from a page, the page type is consulted. If the page type is an array header, then Stasis invokes the array manipulation routines instead of its default slot manipulation code. Special methods that bypass this check are used by the array implementation when it needs to read or modify data stored in the header.

9.7.4 Compression

Stasis' compressed pages are the most complex of the predefined layouts. They borrow ideas from a number of different research projects and commercial products, and use a hybrid of column and row oriented layouts to store data. Each page is partitioned into columns, and records are not allowed to span pages (as in PAX (5)). The columns are encoded using lightweight column compression algorithms (151). Finally, C++ templates are used to generate custom, optimized compression code that is hard coded to schemas defined at compile time.

The compressed page layouts make use of essentially every feature that Stasis' buffer manager and page layout interfaces provide, and are therefore a good example of how complex page layout implementations may be implemented. In addition to the primitives mentioned above, compressed pages provide extremely fast append, iteration and search operations. These map directly to the primitives required by LSM-Trees, which are the intended users of compressed pages (Section 10.2).

9.7.5 Segments

Segment-based recovery is a complex topic (Section 11.4) that is largely outside the scope of this chapter. However, because segments do not include per-page headers, they required a number of modifications to Stasis' page API.

First, Stasis used to track page types and LSNs in the page header. Now, it must track these values separately. Therefore, normal page layouts must copy LSNs to and from this memory as pages are read and written to disk.

Second, although Stasis can automatically infer page types for pages that have headers, it must treat headerless pages specially. The solution is to allow applications that pin pages to optionally pass in a page type. If the pin causes the page to be read into the buffer manager, it is assumed to have the requested type. Otherwise, Stasis checks that the requested type matches the type Stasis expects, and returns an error otherwise.

9.8 Large objects and log-structured storage

Up to this point, this chapter has described techniques that update data in place, with fine-grained control over transaction semantics and durability. Although such techniques address a wide range of workloads, they are typically inappropriate when dealing with large bulk writes, such as those encountered during replication or when working with large amounts of data in bulk.

This section describes an alternative set of techniques. First, it explains how to avoid logging pre- and post-images of large objects without giving up fine-grained transactions. Next, it explains how to build upon techniques for large objects to implement log-structured storage systems that provide many write operations per second, but give up support for fine-grained transactions. This section closes with a discussion of how fine-grained transactions can be reintroduced with little impact upon throughput.

9.8.1 Large objects and Steal/Force transactions

Stasis normally requires at least one physical redo/undo entry for each page that a transaction modifies. The undo is required so that the data that used to be on the page can be restored if the transaction rolls back. The redo is required because Stasis does not force write pages to disk before committing transactions. In general, force writing at commit introduces a number of problems; in particular, it introduces avoidable random writes, and some of the force-written pages may be pinned by other transactions at commit time, which causes transaction commit requests to block on outstanding operations.

None of the drawbacks of forcing pages at commit arise for transactions that write large, newly allocated objects to disk. Such updates are guaranteed to be sequential by Stasis' region allocator (Section 7.3), and any reads to the object would be exposed to uncommitted data. Therefore, we can simply force pages that contain large objects to disk, and avoid recording redo information. If the overwritten pages are newly allocated, and the transaction that freed the space has committed (which the region allocator guarantees), then we know that the old values on the page will never be needed. Therefore, there is no reason to write the undo entry.

In order to allocate and write a large continuous object in Stasis, one need only allocate space for the object, update the pages that contain it, dirty the pages with the LSN of the allocation operation, and finally force write the pages to disk and commit.

9.8.2 Using Stasis to build log-structured storage systems

A wide range of log-structured storage systems exist, but most reduce arbitrary update workloads to sequential writes, leading to suboptimal disk layouts, or to storing data in temporary locations on disk. They then use a background "garbage collection" or "merge" thread to reorganize the data using large sequential writes. Since these writes modify newly allocated pages there is no need to log their contents. Therefore, updates and background tasks proceed at disk speed.

The main drawback to this approach is that, because they bundle large amounts of work into a single Stasis transaction, they are unable to directly support fine-grained transactions. Note that force writes and transactions still provide atomicity and durability. However, they do so for large batches of updates; if the system crashes while updates are in progress, any updates performed since the last batch committed will be lost.

It is possible to reintroduce transactions by logging application requests to a separate *logical log*. In such setups, Stasis provides durability for old writes and the logical log provides durability for more recent updates. Although such techniques are attractive in certain scenarios, they introduce a number of subtle complications. In particular, extra mechanisms are required to ensure that merges and garbage collection do not introduce unacceptable latency variance. Also, it is often difficult for systems based upon logical redo to participate in two-phase commit, especially if transactions' updates are based upon information from external sources that is not included in the logical redo entry.[2]

[2]These issues are similar to those brought up in Section 12.6.2.2, which describes limitations of snapshot-based recovery. The situation there is more dire; in particular, log-structured indexes do not need to periodically quiesce the system, nor do they need to be careful to correctly manage multiple versions of the same page.

Chapter 10

Rose: an example system based on log-structured storage

Rose is a database replication engine for workloads with high volumes of in-place updates. It is designed to provide high-throughput, general purpose transactional replication regardless of database size, query contention and update patterns. In particular, it is designed to run real-time decision support and analytical processing queries against some of today's largest TPC-C style online transaction processing applications. It achieves these goals by making use of a hybrid row-column data layout that simultaneously provides random updates and column-style compression algorithms without resorting to non-sequential I/O.

In order to achieve these goals, it makes use of most of the primitives mentioned in the preceding chapters. Although the majority of the data stored by Rose is managed using log-structured updates (Section 9.8.2), allocation is handled by Stasis' standard, update-in-place allocation routines. Similarly, Rose stores its metadata using standard update-in-place `records`. Rather than use a Stasis transaction for each application update, Rose optimizes for write throughput, and applies many updates per Stasis transaction. It provides snapshot consistency over slightly out of date information. None of these features were anticipated by Stasis' design, yet the only Stasis extensions Rose required were the buffer manager callbacks that enable efficient compression (Section 9.7.4). We now turn to a discussion of the motivation behind Rose, leaving further discussion of Stasis' extensibility for the end of this chapter.

Traditional database replication technologies provide acceptable performance if the application write set fits in RAM or if the storage system is able to update data in place quickly enough to keep up with the replication workload.

Instead of paying the cost of maintaining a perfect data layout, most transaction processing (OLTP) systems allow tables and indexes to become fragmented. This optimizes for small, low-latency reads and writes, but eventually leads to degraded sequential performance. Such systems scale by adding memory and additional drives, which increases the number of I/O operations per second provided by the hardware. Data warehousing technologies introduce latency, giving them time to reorganize data for bulk insertion, and column stores optimize for scan performance at the expense of random access to individual tuples.

Rose combines the best features of these two approaches:

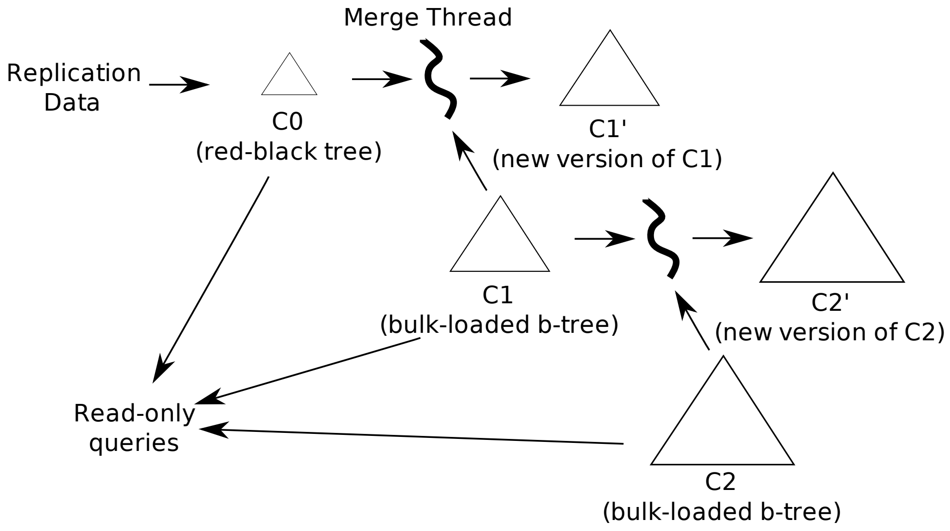


Figure 10.1. The structure of a Rose LSM-tree

- High throughput writes, regardless of update patterns
- Scan performance comparable to bulk-loaded structures
- Low latency lookups and updates

We implemented Rose to bridge the gap between analytical processing systems, which optimize for sequential reads and writes, and transaction processing systems, which optimize for small, scattered, low-latency operations. Rose addresses all three goals.

Rose is based on LSM-trees, which reflect updates immediately without performing disk seeks or resorting to fragmentation. This allows it to provide better write and scan throughput than B-trees. Unlike existing LSM-tree implementations, Rose makes use of compression, further increasing replication and scan performance.

Rose provides extremely high write throughput to applications that perform updates without performing reads, such as replication, append-only, streaming and versioning databases. Replication is a particularly attractive application of LSM-trees because it is common, well-understood and requires complete transactional semantics. Also, we know of no other scalable replication approach that provides real-time analytical queries over seek-bound transaction processing workloads.

10.1 System overview

A Rose replica takes a replication log as input and stores the changes it contains in a *log structured merge* (LSM) tree (105). An LSM-tree is an index that consists of multiple sub-trees called *components* (Figure 10.1). The smallest component, $C0$, is a memory-resident binary search tree that is updated in place. The next-smallest component, $C1$, is a bulk-loaded B-tree. $C0$ is merged with $C1$ as it grows. The merge process consists of index scans and produces a new bulk-loaded version of $C1$ that contains the updates from $C0$. LSM-trees can have arbitrarily many components, though three (two on-disk trees) is generally adequate. All other components are produced by repeated merges with the next smaller component. Therefore, LSM-trees are updated without using random disk I/O.

Lookups are performed by examining tree components, starting with the in-memory component and moving on to progressively larger and out-of-date trees until a match is found. This involves a maximum of two on-disk tree lookups unless a merge is in process, in which case three lookups may be required.

Rose's first contribution is to use compression to increase merge throughput. Compression reduces the amount of sequential I/O required by merges, trading surplus computational power for scarce storage bandwidth. Lookup performance is determined by page cache hit ratios and the number of seeks provided by the storage system. Rose's compression increases the effective size of the page cache. Although it is not fundamental to Rose's design, its implementation makes use of extremely fast, integer-only compression algorithms. The current implementation could be easily extended to support variable-length data types.

The second contribution is the application of LSM-trees to database replication workloads. In order to take full advantage of LSM-trees' write throughput, the replication environment must not force Rose to obtain pre-images of overwritten values. Therefore, we assume that the replication log stores each transaction `begin`, `commit`, and `abort` performed by the master database, along with the pre- and post-images associated with each update. The ordering of these entries must match the order in which they are applied at the database master. Workloads that do not contain transactions or do not update existing data may omit some of the mechanisms described here.

Rose immediately applies updates to $C0$, making them available to queries that do not require a consistent view of the data. Rose supports transactional isolation by delaying the availability of new data for the duration of a few update transactions. It then produces a new, consistent snapshot that reflects the new data. Multiple snapshots can coexist, allowing read-only transactions to run without creating lock contention or being aborted due to concurrent updates.

Long-running transactions do not block Rose's maintenance tasks or index operations. However, long-running read-only queries delay the deallocation of stale data, and long-running updates introduce replication delay. Rose's concurrency control mechanisms are described in Section 10.1.5.

Rose merges tree components in background threads, allowing it to continuously process updates and service index lookup requests. Index lookups minimize the overhead of thread synchronization by latching entire tree components at a time. On-disk tree components are read-only so these latches only block reclamation of space used by deallocated tree components. $C0$ is updated in place, preventing inserts from occurring concurrently with lookups. However, operations on $C0$ are comparatively fast, reducing contention for $C0$'s latch.

Recovery, allocation and atomic updates to Rose's metadata are handled by Stasis; Rose is implemented as a set of custom Stasis page formats and tree structures.

In order to provide inexpensive queries to clients Rose assumes the replication log is made durable by some other mechanism. Rather than commit each replicated transaction independently, it creates one Stasis transaction each time a tree component is created. These transactions generate negligible amounts of log entries.

Redo and undo information for tree component metadata and headers are written to the Stasis log. Data contained in tree components are never written to the log and tree component contents are sequentially forced to the page file at Stasis commit. During recovery, partially written tree components are deallocated and tree headers are brought to a consistent state.

After Stasis recovery completes, Rose is in a consistent state that existed at some point in the past. The replication log must be replayed from that point to finish recovery.

Rose determines where to begin replaying the log by consulting its metadata, which is updated each time a tree merge completes. This metadata contains the timestamp of the last update that is reflected in $C1$, which is simply the timestamp of the last tuple written to $C0$ before the merge began.

10.1.1 Tree merging

Rose's LSM-trees always consist of three components ($C0$, $C1$ and $C2$), as this provides a good balance between insertion throughput and lookup cost. Updates are applied directly to the in-memory tree, and repeated tree merges limit the size of $C0$. These tree merges produce a new version of $C1$ by combining tuples from $C0$ with tuples in the existing version of $C1$. When the merge completes $C1$ is atomically replaced with the new tree and $C0$ is atomically replaced with an empty tree. The process is eventually repeated when $C1$ and $C2$ are merged.

Replacing entire trees at once introduces a number of problems. It doubles the number of bytes used to store each component, which is important for $C0$, as it doubles memory requirements. It is also important for $C2$, as it doubles the amount of disk space used by Rose. Finally, ongoing merges force index probes to access both versions of $C1$, increasing random lookup times.

The original LSM-tree work proposes a more sophisticated scheme that addresses these issues by replacing one sub-tree at a time. This reduces peak storage and memory requirements but adds complexity by requiring in-place updates of tree components.

An alternative approach would partition Rose into multiple LSM-trees and merge a single partition at a time (66). This would reduce the frequency of extra lookups caused by ongoing tree merges. Partitioning also improves write throughput when updates are skewed, as unchanged portions of the tree do not participate in merges and frequently changing partitions can be merged more often. We do not consider these optimizations in the discussion below; including them would complicate the analysis without providing any new insight.

10.1.2 Amortized insertion cost

This section provides an overview of LSM-tree performance. A more thorough analytical discussion (105), and comparisons between LSM-trees and a wide variety of other indexing techniques (66) are available elsewhere.

To compute the amortized LSM-tree insertion cost we consider the cost of comparing the inserted tuple with existing tuples. Each tuple insertion ultimately causes two rounds of I/O operations. One merges the tuple into $C1$; the other merges it into $C2$. Insertions do not initiate more I/O once they reach $C2$.

Write throughput is maximized when the ratio of the sizes of $C1$ to $C0$ is equal to the ratio between $C2$ and $C1$ (105). This ratio is called R, and:

$$size\ of\ tree \approx\ R^2 * |C0|$$

Merges examine an average of R tuples from $C1$ for each tuple of $C0$ they consume. Similarly, R

tuples from $C2$ are examined each time a tuple in $C1$ is consumed. Therefore:

$$insertion\ rate * R(t_{C2} + t_{C1}) \approx\ sequential\ i/o\ cost$$

Where t_{C1} and t_{C2} are the amount of time it takes to read from and write to C1 and C2.

10.1.3 Replication throughput

LSM-trees have different asymptotic performance characteristics than conventional index structures. In particular, the amortized cost of insertion is $O(\sqrt{n}\ log\ n)$ in the size of the data and is proportional to the cost of sequential I/O. In a B-tree, this cost is $O(log\ n)$ but is proportional to the cost of random I/O. This section describes the impact of compression on B-tree and LSM-tree performance.

We use simplified models of each structure's performance characteristics. In particular, we assume that the leaf nodes do not fit in memory, that tuples are accessed randomly with equal probability, and that internal tree nodes fit in RAM. Without a skewed update distribution, reordering and batching I/O into sequential writes only helps if a significant fraction of the tree's data fits in RAM. Therefore, we do not consider B-tree I/O batching here.

In B-trees, each update reads a page and eventually writes it back:

$$cost_{Btree\ update} = 2\ cost_{random\ io}$$

In Rose, we have:

$$cost_{LSMtree\ update} = 2 * 2 * 2 * R * \frac{cost_{sequential\ io}}{compression\ ratio}$$

We multiply by $2R$ because each new tuple is eventually merged into both on-disk components, and each merge involves an average of R comparisons with existing tuples. The second factor of two reflects the fact that the merger must read existing tuples into memory before writing them back to disk.

An update of a tuple is handled as an insertion of the new tuple and a deletion of the old tuple. Deletion is an insertion of a special tombstone tuple that records the deletion of the old version of the tuple, leading to the third factor of two. Updates that do not modify primary key fields avoid this final factor of two. The delete and insert share the same primary key and snapshot number, so the insertion always supersedes the deletion. Therefore, there is no need to insert a tombstone.

The *compression ratio* is $\frac{uncompressed\ size}{compressed\ size}$, so improved compression leads to less expensive LSM-tree updates. For simplicity, we assume the compression ratio is the same throughout each component of the LSM-tree; Rose addresses this at runtime by reasoning in terms of the number of pages used by each component.

Our test hardware's hard drive is a 7200RPM, 750 GB Seagate Barracuda ES. Third party benchmarks (133) report random access times of 12.3/13.5 msec (read/write) and 44.3-78.5 megabytes/sec sustained throughput. Timing `dd if=/dev/zero of=file; sync` on an empty ext3 file system suggests our test hardware provides 57.5 megabytes/sec of storage bandwidth, but running a similar test via Stasis' buffer manager produces a multimodal distribution ranging from 22 to 47 megabytes/sec.

Assuming a fixed hardware configuration and measuring cost in disk time, we have:

$$cost_{sequential} = \frac{|tuple|}{78.5MB/s} = 12.7\ |tuple|\ \ nsec/tuple\ (min)$$

$$cost_{sequential} = \frac{|tuple|}{44.3MB/s} = 22.6\ |tuple|\ \ nsec/tuple\ (max)$$

and

$$cost_{random} = \frac{12.3 + 13.5}{2} = 12.9\ msec/tuple$$

Assuming $44.3mb/s$ sequential bandwidth:

$$2\ cost_{random} \approx 1,000,000 \frac{cost_{sequential}}{|tuple|}$$

yields:

$$\frac{cost_{LSMtree\ update}}{cost_{Btree\ update}} = \frac{2 * 2 * 2 * R * cost_{sequential}}{compression\ ratio * 2 * cost_{random}}$$
$$\approx \frac{R * |tuple|}{250,000 * compression\ ratio}$$

If tuples are 100 bytes and we assume a compression ratio of 4, which is lower than we expect to see in practice, but numerically convenient, then the LSM-tree outperforms the B-tree when:

$$R < \frac{250,000 * compression\ ratio}{|tuple|}$$

$$R < 10,000$$

A 750 GB tree with a 1GB $C0$ would have an R of $\sqrt{750} \approx 27$. If tuples are 100 bytes such a tree's sustained insertion throughput will be approximately 8000 tuples/sec or 800 kilobytes/sec.

Our hard drive's average access time allows the drive to deliver 83 I/O operations/sec. Therefore, we can expect an insertion throughput of 41.5 tuples/sec from a B-tree that does not cache leaf pages. With 1 GB of RAM, Rose should outperform the B-tree by more than two orders of magnitude. Increasing Rose's system memory to cache 10 GB of tuples would increase write performance by a factor of $\sqrt{10}$.

Increasing memory another ten fold to 100 GB would yield an LSM-tree with an R of $\sqrt{750/100} = 2.73$ and a throughput of 81,000 tuples/sec. In contrast, the B-tree would cache less than 100 GB of leaf pages in memory and would write fewer than $\frac{41.5}{1-(100/750)} = 47.9$ tuples/sec. Increasing memory further yields a system that is no longer disk bound.

Assuming CPUs are fast enough to allow Rose to make use of the bandwidth supplied by the disks, we conclude that Rose will provide significantly higher throughput than a seek-bound B-tree. Our experiments (Section 10.3) show that this is indeed the case.

10.1.4 Indexing

Our analysis ignores the cost of bulk-loading and storing LSM-trees' internal nodes. Each time the merge process fills a page it inserts an entry into the rightmost position in the tree, allocating additional internal nodes if necessary. Our prototype does not compress internal tree nodes.

The space overhead of building these tree nodes depends on the number of tuples that fit in each page. If tuples take up a small fraction of each page then Rose gets good fan-out on internal nodes, reducing the fraction of storage used by those nodes. Therefore, as page size increases, tree overhead decreases. Similarly, as tuple sizes increase, the fraction of storage dedicated to internal tree nodes increases.

Rose pages are 4KB. Workloads with larger tuples would save disk and page cache space by using larger pages for internal nodes. For very large trees, larger internal tree nodes also avoid seeks during index lookups.

10.1.5 Isolation

Rose handles two types of transactions: updates from the master and read-only queries from clients. Rose has no control over the order or isolation of updates from the master database. Therefore, it must apply these updates in an order consistent with the master database in a way that isolates queries from concurrent modifications.

LSM-tree updates do not immediately remove pre-existing data from the tree. Instead, they provide a simple form of versioning where data from the newest component ($C0$) is always the most up-to-date. We extend this idea to provide snapshots. Each transaction is assigned to a snapshot, and no more than one version of each tuple exists within a snapshot.

To lookup a tuple as it existed at some point in time, we examine all snapshots that were taken before that point and return the most recent version of the tuple that we find. This is inexpensive because all versions of the same tuple are stored in adjacent positions in the LSM-tree. Furthermore, Rose's versioning is meant to be used for transactional consistency and not time travel, limiting the number of snapshots.

To insert a tuple, Rose first determines to which snapshot it should belong. At any given point in time up to two snapshots may accept new updates. One snapshot accepts new transactions. If the replication log may contain interleaved transactions, such as when there are multiple master databases, then a second, older snapshot waits for any pending transactions that it contains to complete. Once all of those transactions are complete, the older snapshot stops accepting updates, the first snapshot stops accepting new transactions, and a snapshot for new transactions is created.

Rose examines the timestamp and transaction ID associated with the tuple insertion and marks the tuple with the appropriate snapshot ID. It then inserts the tuple into $C0$, overwriting any tuple that matches on primary key and has the same snapshot ID.

The merge thread uses a list of currently accessible snapshots to decide which versions of the tuple in $C0$ and $C1$ are accessible to transactions. Such versions are the most recent copy of the tuple that existed before the end of an accessible snapshot. After finding all such versions of a tuple (one may exist for each accessible snapshot), the merge process writes them to the new version of $C1$. Any other versions of the tuple are discarded. If two matching tuples from the same snapshot are encountered then one must be from $C0$ and the other from $C1$. The version from $C0$ is more recent, so the merger discards the version from $C1$. Merges between $C1$ and $C2$ work the same way as merges between $C0$ and $C1$.

Rose handles tuple deletion by inserting a tombstone. Tombstones record the deletion event and are handled similarly to insertions. A tombstone will be kept if it is the newest version of a tuple in at least one accessible snapshot. Otherwise, the tuple was reinserted before next accessible snapshot was taken and the tombstone is deleted. Unlike insertions, tombstones in $C2$ are deleted if they are the oldest remaining reference to a tuple.

Format	Compression	Page count
PFOR	1.96x	2494
PFOR + tree	1.94x	+80
RLE	3.24x	1505
RLE + tree	3.22x	+21

Table 10.1. Compression ratios and index overhead - five columns (20 bytes/column)

Format	Compression	Page count
PFOR	1.37x	7143
PFOR + tree	1.17x	8335
RLE	1.75x	5591
RLE + tree	1.50x	6525

Table 10.2. Compression ratios and index overhead - 100 columns (400 bytes/column)

Rose's snapshots have minimal performance impact, and provide transactional concurrency control without rolling back transactions or blocking the merge and replication processes. However, long-running updates prevent queries from accessing the results of recent transactions, leading to stale results. Long-running queries increase Rose's disk footprint by increasing the number of accessible snapshots.

10.1.6 Parallelism

Rose operations are concurrent; readers and writers work independently, avoiding blocking, deadlock and livelock. Index probes must latch $C0$ to perform a lookup, but the more costly probes into $C1$ and $C2$ are against read-only trees. Beyond locating and pinning tree components against deallocation, probes of these components do not interact with the merge processes.

Each tree merge runs in a separate thread. This allows our prototype to exploit two to three processor cores while inserting and recompressing data during replication. Remaining cores could be exploited by range partitioning a replica into multiple LSM-trees, allowing more merge processes to run concurrently. Therefore, we expect the throughput of Rose replication to increase with memory size, compression ratios and I/O bandwidth for the foreseeable future.

10.2 Row compression

Rose stores tuples in a sorted, append-only format. This simplifies compression and provides a number of new opportunities for optimization. Compression reduces sequential I/O, which is Rose's primary bottleneck. It also increases the effective size of the buffer pool, allowing Rose to service larger read sets without resorting to random I/O.

Row-oriented database compression techniques must cope with random, in-place updates and provide efficient random access to compressed tuples. In contrast, compressed column-oriented database layouts focus on high-throughput sequential access, and do not provide in-place updates or efficient random access. Rose never updates data in place, allowing it to use append-only compression techniques from the column database literature. Also, Rose's tuples never span pages and are stored in sorted order. We modified column compression techniques to provide an index over each page's contents and efficient random access within pages.

Rose's compression format is straightforward. Each page is divided into a header, a set of compressed segments and a single exception section (Figure 10.2). There is one compressed segment per column and each such segment may be managed by a different compression algorithm, which we call a *compressor*. Some compressors cannot directly store all values that a column may take on. Instead, they store such values in the exception section. We call Rose's compressed page format the *multicolumn* format, and have implemented it to support efficient compression, decompression and random lookups by slot id and by value.

10.2.1 Compression algorithms

The Rose prototype provides three compressors. The first, *NOP*, simply stores uncompressed integers. The second, *RLE*, implements run length encoding, which stores values as a list of distinct values and repetition counts. The third, *PFOR*, or patched frame of reference, stores values as a single per-page base offset and an array of deltas (151). The values are reconstructed by adding the offset and the delta or by traversing a pointer into the exception section of the page. Currently, Rose only supports integer data although its page formats could be easily extended with support for strings and other types. Also, the compression algorithms that we implemented would benefit from a technique known as *bit-packing*, which represents integers with lengths other than 8, 16, 32 and 64 bits. Bit-packing would increase Rose's compression ratios, and can be implemented with a modest performance overhead (151).

We chose these techniques because they are amenable to optimization; our implementation makes heavy use of C++ templates, static code generation and g++'s optimizer to keep Rose from becoming CPU-bound. By hard coding table formats at compilation time, this implementation removes length and type checks, and allows the compiler to inline methods, remove temporary variables and optimize loops.

Rose includes a second, less efficient implementation that uses dynamic method dispatch to support runtime creation of new table schemas. A production-quality system could use runtime code generation to support creation of new schemas while taking full advantage of the compiler's optimizer.

10.2.2 Comparison with other approaches

Like Rose, the PAX (5) page format stores tuples in a way that never spans multiple pages and partitions data within pages by column. Partitioning pages by column improves processor cache locality for queries that do not need to examine every column within a page. In particular, many queries filter based on the value of a few attributes. If data is clustered into rows then each column from the page will be brought into cache, even if no tuples match. By clustering data into columns, PAX ensures that only the necessary columns are brought into cache.

Rose's use of lightweight compression algorithms is similar to approaches used in column databases (151). Rose differs from that work in its focus on small, low-latency updates and efficient random reads on commodity hardware. Existing work targeted RAID and used page sizes ranging from 8-32MB. Rather than use large pages to get good sequential I/O performance, Rose relies upon automatic prefetch and write coalescing provided by Stasis' buffer manager and the

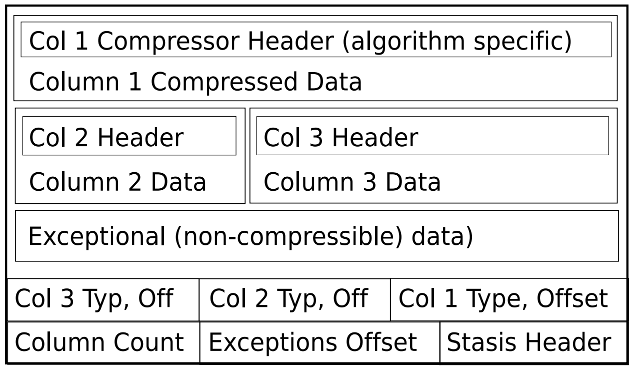


Figure 10.2. Multicolumn page format. Many compression algorithms can coexist on a single page. Tuples never span multiple pages.

underlying operating system. This reduces the amount of data handled by Rose during index probes.

10.2.3 Tuple lookups

Because we expect index lookups to be a frequent operation, our page format supports efficient lookup by tuple value. The original "patched" compression algorithms store exceptions in a linked list and perform scans of up to 128 tuples in order to materialize compressed tuples (151). This reduces the number of branches required during compression and sequential decompression. We investigated a number of implementations of tuple lookup by value including per-column range scans and binary search.

The efficiency of random access within a page depends on the format used by column compressors. Rose compressors support two access methods. The first looks up a value by slot id. This operation is $O(1)$ for frame of reference columns and $O(log\ n)$ in the number of runs of identical values on the page for run length encoded columns.

The second operation is used to look up tuples by value and is based on the assumption that the tuples are stored in the page in sorted order. The simplest way to provide access to tuples by value would be to perform a binary search, materializing each tuple that the search needs to examine. Doing so would materialize many extra column values, potentially performing additional binary searches.

To lookup a tuple by value, the second operation takes a range of slot ids and a value, and returns the offset of the first and last instance of the value within the range. This operation is $O(log\ n)$ in the number of slots in the range for frame of reference columns and $O(log\ n)$ in the number of runs on the page for run length encoded columns. The multicolumn implementation uses this method to look up tuples by beginning with the entire page in range and calling each compressor's implementation to narrow the search until the correct tuple(s) are located or the range is empty. Partially matching tuples are only partially decompressed during the search, reducing the amount of data brought into processor cache.

Format	Ratio	Compress	Decompress
PFOR - 1 column	3.96x	547 mb/s	2959 mb/s
PFOR - 10 column	3.86x	256	719
RLE - 1 column	48.83x	960	1493 (12%)
RLE - 10 column	47.60x	358 (9%)	659 (7%)

Table 10.3. Compressor throughput—Random data Mean of 5 runs, $\sigma < 5\%$, except where noted. RLE stands for "run length encoding" compression, while PFOR stands for "patched frame of reference" compression.

10.2.4 Multicolumn computational overhead

Our multicolumn page format introduces additional computational overhead. Rose compresses each column in a separate buffer then uses `memcpy()` to gather this data into a single page buffer before writing it to disk. This happens once per page allocation.

Bit-packing is typically implemented as a post-processing technique that makes a pass over the compressed data (151). Rose's gathering step can be performed for free by such a bit-packing implementation, so we do not see this as a significant disadvantage compared to other approaches.

Also, Rose needs to translate tuple insertions into calls to appropriate page formats and compression implementations. Unless we hard code the Rose executable to support a predefined set of page formats and table schemas, each tuple compression and decompression operation must execute an extra `for` loop over the columns. The `for` loop's body contains a `switch` statement that chooses between column compressors, since each column can use a different compression algorithm and store a different data type.

This form of multicolumn support introduces significant overhead; these variants of our compression algorithms run significantly slower than versions hard coded to work with single column data. Table 10.3 compares the performance of a single column page layout that has been hard coded to a single compression algorithm with analogous page layouts that support multiple columns, and that are not hard coded to particular compression algorithm.

10.2.5 The `append()` operation

Rose's compressed pages provide a `tupleAppend()` operation that takes a tuple as input and returns `false` if the page does not have room for the new tuple. `tupleAppend()` consists of a dispatch routine that calls `append()` on each column in turn. `append()` has the following signature:

```
void append(COL_TYPE value, int* exception_offset,
    void* exceptions_base, void* column_base,
    int* freespace)
```

where `value` is the value to be appended to the column, `exception_offset` is a pointer to the offset of the first free byte in the exceptions region, and `exceptions_base` and `column_base` point to page-sized buffers used to store exceptions and column data as the page is being written. One copy of these buffers exists for each LSM-tree component; they do not significantly increase Rose's memory requirements.

`freespace` is a pointer to the number of free bytes remaining on the page. The multicolumn format initializes these values when the page is allocated. As `append()` implementations are called they update this data accordingly. Each time a column value is written to a page the column compressor must allocate space to store the new value. In a naive allocation approach the compressor would check `freespace` to decide if the page is full and return an error code otherwise. Then its caller would check the return value and behave appropriately. This would lead to two branches per column value, greatly decreasing compression throughput.

We avoid these branches by reserving a portion of the page approximately the size of a single incompressible tuple for speculative allocation. Instead of checking for freespace, compressors decrement the current freespace count and append data to the end of their segment. Once an entire tuple has been written to a page, it checks the value of freespace and decides if another tuple may be safely written. This also simplifies corner cases; if a compressor cannot store more tuples on a page, but has not run out of space[1] it simply sets `freespace` to -1 and returns.

The original PFOR implementation (151) assumes it has access to a buffer of uncompressed data and is able to make multiple passes over the data during compression. This allows it to remove branches from loop bodies, improving compression throughput. We opted to avoid this approach in Rose because it would increase the complexity of the `append()` interface and add a buffer to Rose's merge threads.

10.2.6 Buffer manager interface extensions

In the process of implementing Rose, we added a new API to Stasis' buffer manager implementation. It consists of four functions: `pageLoaded()`, `pageFlushed()`, `pageEvicted()`, and `cleanupPage()`.

Stasis supports custom page layouts. These layouts control the byte-level format of pages and must register callbacks that will be invoked by Stasis at appropriate times. The first three are invoked by the buffer manager when it loads an existing page from disk, writes a page to disk, and evicts a page from memory.

The fourth is invoked by page allocation routines immediately before a page is reformatted to use a different layout. This allows the page's old layout's implementation to free any in-memory resources that it associated with the page during initialization or when `pageLoaded()` was called.

We register implementations for these functions because Stasis maintains background threads that control eviction of Rose's pages from memory. As we mentioned above, multicolumn pages are split into a number of temporary buffers while they are being created and are then packed into a contiguous buffer before being flushed. Multicolumn's `pageFlushed()` callback guarantees that this happens before the page is written to disk. `pageLoaded()` parses the page headers and associates statically generated compression code with each page as it is read into memory. `pageEvicted()` and `cleanupPage()` free memory that is allocated by `pageLoaded()`.

`pageFlushed()` could be safely executed in a background thread with minimal impact on system performance. However, the buffer manager was written under the assumption that the cost of in-memory operations is negligible. Therefore, it blocks all buffer management requests while `pageFlushed()` is being executed. In practice, copying compression buffers into the buffer manager's page from within `pageFlushed()` would block multiple Rose threads.

Also, copying the data into its on-disk layout reduces Rose's memory utilization by freeing up

[1]This can happen when a run length encoded page stores a very long run of identical tuples.

Column Name	Compression Format	Key
Longitude	RLE	*
Latitude	RLE	*
Timestamp	PFOR	*
Weather conditions	RLE	
Station ID	RLE	
Elevation	RLE	
Temperature	PFOR	
Wind Direction	PFOR	
Wind Speed	PFOR	
Wind Gust Speed	RLE	

Table 10.4. Weather data schema

temporary compression buffers. For these reasons, the merge threads explicitly pack pages instead of waiting for `pageFlushed()` to be called.

10.2.7 Storage overhead

The multicolumn page format is similar to the format of existing column-wise compression formats. The algorithms we implemented have page formats that can be divided into two sections. The first section is a header that contains an encoding of the size of the compressed region and perhaps a piece of uncompressed data. The second section typically contains the compressed data.

A multicolumn page contains this information in addition to metadata describing the position and type of each column. The type and number of columns could be encoded in the "page type" field or be explicitly represented using a few bytes per page column. If we use 16 bits to represent the page offset and 16 bits for the column compressor type then the additional overhead for each column is four bytes plus the size of the compression format headers.

A frame of reference column header consists of a single uncompressed value and two bytes to record the number of encoded rows. Run length encoding headers consist of a two byte count of compressed blocks. Therefore, in the worst case (frame of reference encoding 64-bit integers, and 4KB pages) our prototype's multicolumn format uses 14 bytes, or 0.35% of the page to store each column header. Bit-packing and storing a table that maps page types to lists of column and compressor types would reduce the size of the page headers.

In cases where the data does not compress well and tuples are large, additional storage is wasted because Rose does not split tuples across pages. Table 10.2 illustrates this; it was generated in the same way as Table 10.1, except that 400 byte tuples were used instead of 20 byte tuples. Larger pages would reduce the impact of this problem. We chose 4KB pages because they are large enough for the schemas we use to benchmark Rose, but small enough to illustrate the overheads of our page format.

10.3 Evaluation

10.3.1 Raw write throughput

To evaluate Rose's raw write throughput, we used it to index weather data. The data set ranges from May 1, 2007 to Nov 2, 2007 and contains readings from ground stations around the world (101). Our implementation assumes these values will never be deleted or modified. Therefore, for this experiment the tree merging threads do not perform versioning or snapshotting. This data is approximately $1.3GB$ when stored in an uncompressed tab delimited file. We duplicated the data by changing the date fields to cover ranges from 2001 to 2009, producing a 12GB ASCII dataset that contains approximately 132 million tuples.

Duplicating the data should have a limited effect on Rose's compression ratios. We index on geographic position, placing all readings from a particular station in a contiguous range. We then index on date, separating duplicate versions of the same tuple from each other.

Rose only supports integer data types. We store ASCII columns for this benchmark by packing each character into 5 bits (the strings consist of A-Z, "+," "-" and "*") and storing them as integers. Floating point columns in the original data set are always represented with two digits of precision; we multiply them by 100, yielding an integer. The data source uses nonsensical readings such as -9999.00 to represent NULL. Our prototype does not understand NULL, so we leave these fields intact.

We represent each integer column as a 32-bit integer, even when we could use a 16-bit value. The "weather conditions" field is packed into a 64-bit integer. Table 10.4 lists the columns and compression algorithms we assigned to each column. The "Key" column indicates that the field was used as part of a B-tree primary key.

In this experiment, we randomized the order of the tuples and inserted them into the index. We compare Rose's performance with the MySQL InnoDB storage engine. We chose InnoDB because it has been tuned for bulk load performance, and we avoid the overhead of SQL insert statements and MySQL transactions by using MySQL's bulk load interface. Data is loaded 100,000 tuples at a time so that MySQL inserts values into its tree throughout the run and does not sort and bulk load the data all at once.

InnoDB's buffer pool size was 2GB and its log file size was 1GB. We enabled InnoDB's doublewrite buffer, which writes a copy of each updated page to a sequential log. The doublewrite buffer increases the total amount of I/O performed, but decreases the frequency with which InnoDB calls `fsync()` while writing dirty pages to disk. This increases replication throughput for this workload.

We compiled Rose's C components with "-O2", and the C++ components with "-O3". The later compiler flag is crucial, as compiler inlining and other optimizations improve Rose's compression throughput significantly. Rose was set to allocate $1GB$ to $C0$ and another $1GB$ to its buffer pool. In this experiment, Rose's buffer pool is essentially wasted once its page file size exceeds 1GB; Rose accesses the page file sequentially and evicts pages using LRU, leading to a cache hit rate near zero.

Our test hardware has two dual core 64-bit 3GHz Xeon processors with 2MB of cache (Linux reports 4 CPUs) and 8GB of RAM. We disabled the swap file and unnecessary system services. Data sets large enough to become disk bound on this system are unwieldy, so we `mlock()` 5.25GB of RAM to prevent it from being used by experiments. The remaining 750MB is used to cache

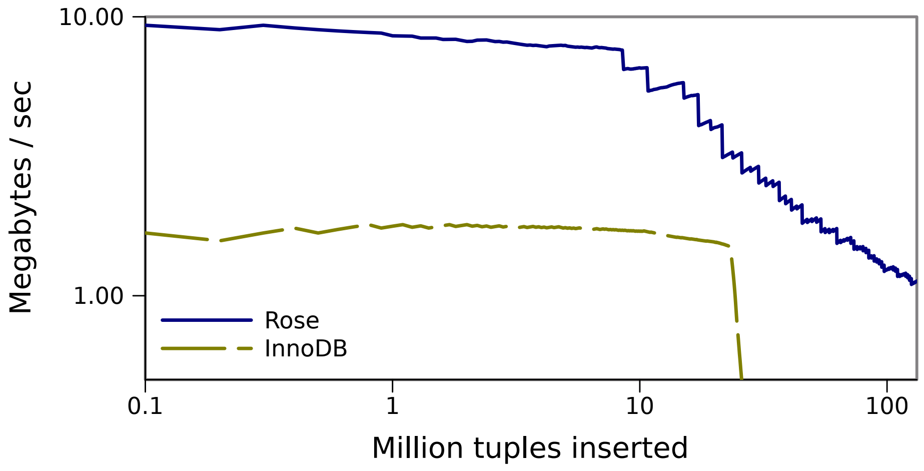


Figure 10.3. Mean insertion throughput (log-log). We terminated the InnoDB experiment before it completed; throughput was 20kb/s ($\frac{1}{57}$th of Rose's) after 50 million insertions.

binaries and to provide Linux with enough page cache to prevent it from unexpectedly evicting portions of the Rose binary. We monitored Rose throughout the experiment, confirming that its resident memory size was approximately 2GB.

All software used during our tests was compiled for 64-bit architectures. We used a 64-bit Ubuntu Gutsy (Linux 2.6.22-14-generic) installation and its prebuilt MySQL package (5.0.45-Debian_1ubuntu3). All page files and logs were stored on a dedicated drive. The operating system, executables and input files were stored on another drive.

10.3.2 Comparison with conventional techniques

Rose provides roughly 4.7 times more throughput than InnoDB at the beginning of the experiment (Figure 10.3). InnoDB's throughput remains constant until it starts to perform random I/O, which causes throughput to drop sharply.

Rose's performance begins to fall off earlier due to merging and because its page cache is half the size of InnoDB's. However, Rose does not fall back to random I/O and maintains significantly higher throughput than InnoDB throughout the run. InnoDB's peak write throughput was 1.8 mb/s and dropped to 20 kb/s after 50 million tuples were inserted. Note that we did not wait for the InnoDB trend line in Figure 10.3 to level off before terminating the experiment; its performance was dropping steadily at the end of the run. In contrast, Rose maintained an average throughput of 1.13 mb/sec over the entire 132 million tuple dataset.

Rose merged $C0$ and $C1$ 59 times and merged $C1$ and $C2$ 15 times. At the end of the run (132 million tuple insertions) $C2$ took up 2.8GB and $C1$ was 250MB. The actual page file was 8.0GB, and the minimum possible size was 6GB. InnoDB used 5.3GB after 53 million tuple insertions.

10.3.3 Comparison with analytical model

In this section we measure update latency and compare measured write throughput with the analytical model's predicted throughput.

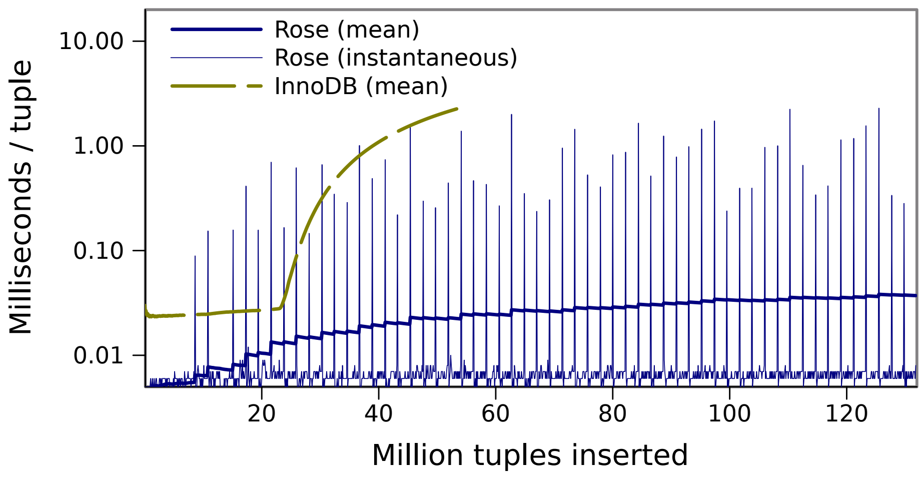


Figure 10.4. Tuple insertion time (log). "Instantaneous" is the mean over 100,000 tuples. Rose's cost increases with $\sqrt{n}$; InnoDB's increases linearly; as expected, InnoDB's "instantaneous" performance is a smooth line (not shown).

Figure 10.4 shows tuple insertion times for Rose and InnoDB. The "Rose (instantaneous)" line reports insertion times averaged over 100,000 insertions, while the other lines are averaged over the entire run. The periodic spikes in instantaneous tuple insertion times occur when an insertion blocks waiting for a tree merge to complete. This happens when one copy of $C0$ is full and the other one is being merged with $C1$. Admission control would provide consistent insertion times.

Figure 10.5 compares our prototype's performance to that of an ideal uncompressed LSM-tree. The cost of an insertion is $4R$ times the number of bytes written. We can approximate an optimal value for R by taking $\sqrt{\frac{|C2|}{|C0|}}$, where tree component size is measured in number of tuples. This factors out compression and memory overheads associated with $C0$.

We use this value to generate Figure 10.5 by multiplying Rose's replication throughput by $1 + 4R$. The additional 1 is the cost of reading the inserted tuple from $C1$ during the merge with $C2$.

Rose achieves 2x compression on the real data set. For comparison, we ran the experiment with compression disabled, and with run length encoded synthetic data sets that lead to 4x and 8x compression ratios. Initially, all runs exceed optimal throughput as they populate memory and produce the initial versions of $C1$ and $C2$. Eventually, each run converges to a constant effective disk utilization roughly proportional to its compression ratio. This shows that Rose continues to be I/O bound with higher compression ratios.

The analytical model predicts that, given the hard drive's 57.5 mb/s write bandwidth, an ideal Rose implementation would perform about twice as fast as our prototype. We believe the difference is largely due to Stasis' buffer manager, which delivers between 22 and 47 mb/s of sequential bandwidth on our test hardware. The remaining difference in throughput is probably due to Rose's imperfect overlapping of computation with I/O and coarse-grained control over component sizes and R.

Despite these limitations, compression allows Rose to deliver significantly higher throughput

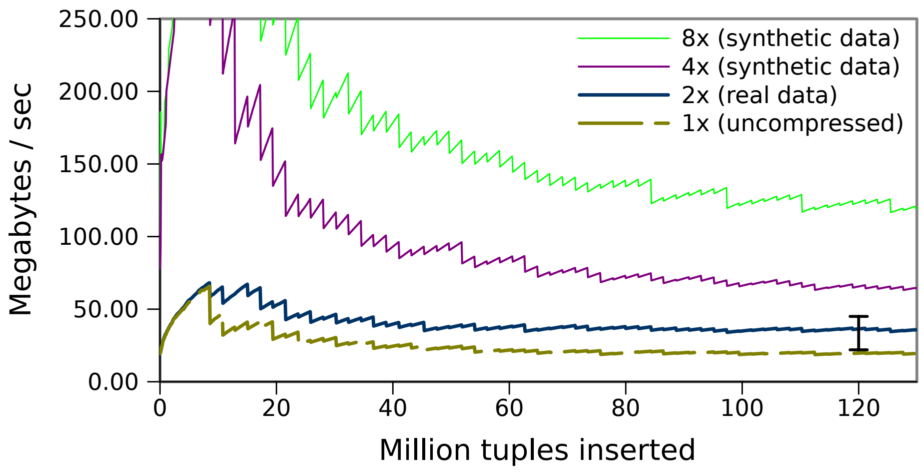


Figure 10.5. Disk bandwidth an uncompressed LSM-tree would require to match Rose's throughput. The trend lines were calculated by plugging Rose's actual throughput into the analytical model of its performance, and calculating the raw (uncompressed) number of bytes being stored in the buffer manager. In this experiment, Stasis' buffer manager provided 22-45 mb/s. The leveling of the trend lines indicates that Rose is behaving in accordance with the analytical model, and the lines' relationship to the compression ratio shows that compression improves throughput.

than would be possible with an uncompressed LSM-tree. We expect Rose's throughput to increase with the size of RAM and storage bandwidth for the foreseeable future.

10.3.4 TPC-C / H

TPC-H is an analytical processing benchmark that targets periodically bulk-loaded data warehousing systems. In particular, compared to TPC-C, it de-emphasizes transaction processing and rollback and allows database vendors to permute the dataset off-line. In real-time database replication environments faithful reproduction of transaction processing schedules is important and there is no opportunity to sort data before making it available to queries. Therefore, we insert data in chronological order.

Our Rose prototype is far from a complete storage engine implementation. Rather than implement relational algebra operations and attempt to process and appropriately optimize SQL queries on top of Rose, we chose a small subset of the TPC-H and C benchmark queries and wrote custom code to invoke appropriate Rose tuple modifications, table scans and index lookup requests. For simplicity, updates and queries are performed by a single thread.

When modifying TPC-H and C for our experiments, we follow an existing approach (110; 62) and start with a pre-computed join and projection of the TPC-H dataset. We use the schema described in Table 10.5, and populate the table by using a scale factor of 30 and following the random distributions dictated by the TPC-H specification. The schema for this experiment is designed to have poor update locality.

Updates from customers are grouped by order id, but the index is sorted by product and date.

Column Name	Compression Format	Data type
Part # + Supplier	RLE	int32
Year	RLE	int16
Week	NONE	int8
Day of week	NONE	int8
Order number	NONE	int64
Quantity	NONE	int8
Delivery Status	RLE	int8

Table 10.5. TPC-C/H schema

This forces the database to permute these updates into an order that would provide suppliers with inexpensive access to lists of orders to be filled and historical sales information for each product.

We generate a dataset containing a list of product orders, and insert tuples for each order (one for each part number in the order), then add a number of extra transactions. The following updates are applied in chronological order:

- Following TPC-C's lead, 1% of orders are immediately canceled and rolled back. This is handled by inserting a tombstone for the order.
- Remaining orders are delivered in full within the next 14 days. The order completion time is chosen uniformly at random between 0 and 14 days after the order was placed.
- The status of each line item is changed to "delivered" at a time chosen uniformly at random before the order completion time.

The following read-only transactions measure the performance of Rose's access methods:

- Every 100,000 orders we initiate a table scan over the entire data set. The per-order cost of this scan is proportional to the number of orders processed so far.
- 50% of orders are checked with order status queries. These are simply index probes that lookup a single line item (tuple) from the order. Orders that are checked with status queries are checked 1, 2 or 3 times with equal probability. Line items are chosen randomly with equal probability.
- Order status queries happen with a uniform random delay of 1.3 times the order processing time. For example, if an order is fully delivered 10 days after it is placed, then order status queries are timed uniformly at random within the 13 days after the order is placed.

This dataset is not easily compressible using the algorithms provided by Rose. Many columns fit in a single byte, rendering Rose's version of PFOR useless. These fields change frequently enough to limit the effectiveness of run length encoding. Both of these issues would be addressed by bit packing. Also, occasionally re-evaluating and modifying compression strategies is known to improve compression of TPC-H data. TPC-H dates are clustered during weekdays, from 1995-2005, and around Mother's Day and the last few weeks of each year.

Order status queries have excellent temporal locality and generally succeed after accessing $C0$. These queries simply increase the amount of CPU time between tuple insertions and have

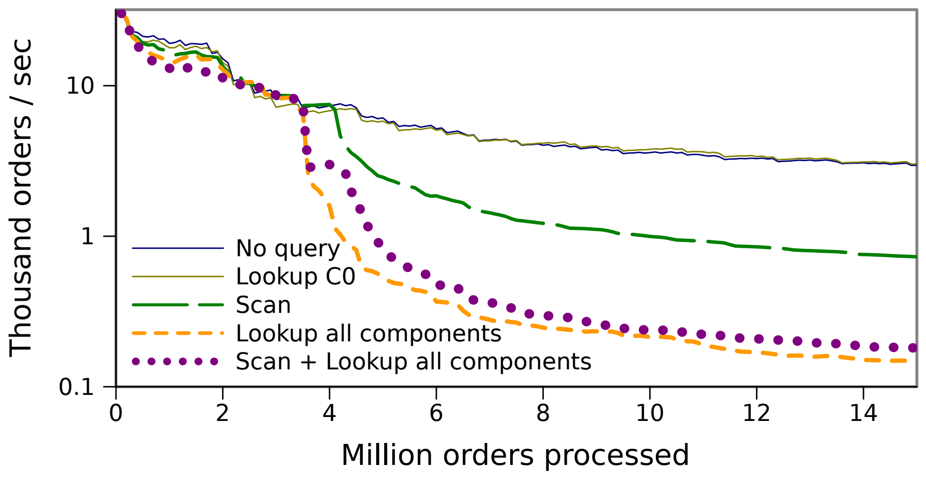


Figure 10.6. Rose TPC-C/H order throughput

minimal impact on replication throughput. Rose overlaps their processing with the asynchronous I/O performed by merges.

We force Rose to become seek bound by running a second set of experiments with a different version of the order status query. In one set of experiments, which we call "Lookup C0," the order status query only examines $C0$. In the other, which we call "Lookup all components," we force each order status query to examine every tree component. This keeps Rose from exploiting the fact that most order status queries can be serviced from $C0$. Finally, Rose provides versioning for this test; though its garbage collection code is executed, it never collects overwritten or deleted tuples.

Figure 10.6 plots the number of orders processed by Rose per second against the total number of orders stored in the Rose replica. For this experiment, we configure Rose to reserve 1GB for the page cache and 2GB for $C0$. We `mlock()` 4.5GB of RAM, leaving 500MB for the kernel, system services, and Linux's page cache.

The cost of searching $C0$ is negligible, while randomly accessing the larger components is quite expensive. The overhead of index scans increases as the table increases in size, leading to a continuous downward slope throughout runs that perform scans.

Surprisingly, periodic table scans improve lookup performance for $C1$ and $C2$. The effect is most pronounced after 3 million orders are processed. That is approximately when Stasis' page file exceeds the size of the buffer pool, which is managed using LRU. After each merge, half the pages it read become obsolete. Index scans rapidly replace these pages with live data using sequential I/O. This increases the likelihood that index probes will be serviced from memory. A more sophisticated page replacement policy would further improve performance by evicting obsolete pages before accessible pages.

We use InnoDB to measure the cost of performing B-tree style updates. In this experiment, we limit InnoDB's page cache to $1GB$, and `mlock()` $6GB$ of RAM. InnoDB does not perform any order status queries or scans (Figure 10.7). Although Rose has a total of $3GB$ for this experiment, it only uses $1GB$ for queries, so this setup causes both systems to begin performing random I/O at the same time.

InnoDB's performance degrades rapidly once it begins using random I/O to update its B-tree.

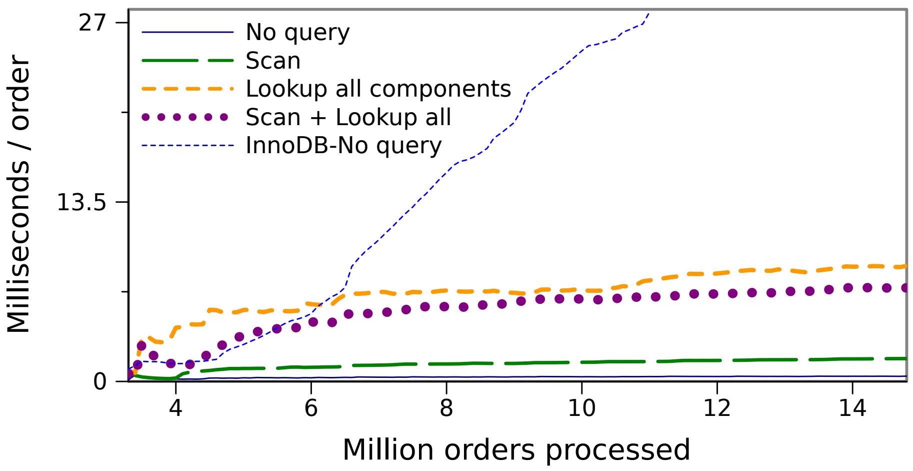


Figure 10.7. Rose and InnoDB TPC-C/H order times, averaged starting after 3.3 million orders. Drive access time is ∼13.5ms. Reading data during updates dominates the cost of queries.

This is because the sort order of its B-tree does not match the ordering of the updates it processes. Ignoring repeated accesses to the same tuple within the same order, InnoDB performs two B-tree operations per line item (a read and a write), leading to many page accesses per order.

In contrast, Rose performs an index probe 50% of the time. Rose's index probes read $C1$ and $C2$, so it accesses one page per order on average. However, by the time the experiment concludes, pages in $C1$ are accessed R times more often (~ 6.6) than those in $C2$, and the page file is 3.9GB. This allows Rose to keep $C1$ cached in memory, so each order uses approximately half a disk seek. At larger scale factors, Rose's access time should double, but remain well below the time a B-tree would spend applying updates.

After terminating the InnoDB run, we allowed MySQL to quiesce, then performed an index scan over the table. At this point, it had processed 11.8 million orders, and index scans took 19 minutes and 20 seconds. Had we performed index scans during the InnoDB run this would translate to an 11.5 ms cost per order. The overhead of Rose scans at that point in the run was approximately 2.2 ms per order.

10.4 Related Work

10.4.1 LSM-trees

The original LSM-tree work (105) provides a more detailed analytical model than the one presented above. It focuses on update intensive OLTP (TPC-A) workloads and hardware provisioning for steady state workloads.

LHAM is an adaptation of LSM-trees for hierarchical storage systems (100). It stores higher numbered components on archival media such as CD-R or tape drives, and scales LSM-trees beyond the capacity of high-performance storage media. It also supports efficient time travel queries over archived, historical data (100).

Partitioned exponential files are similar to LSM-trees, except that they range partition data into smaller indexes (66). This solves a number of issues that are left unaddressed by Rose, most notably skewed update patterns and merge storage overhead.

Rose is optimized for uniform random insertion patterns and does not attempt to take advantage of skew in the replication workload. If most updates are to a particular partition then partitioned exponential files merge that partition frequently, skipping merges of unmodified partitions. Also, smaller merges duplicate smaller components, wasting less space. Multiple merges can run in parallel, improving concurrency. Partitioning a Rose replica into multiple LSM-trees would enable similar optimizations.

Partitioned exponential files avoid a few other pitfalls of LSM-tree implementations. Rose addresses these problems in different ways. One issue is page file fragmentation. Partitioned exponential files make use of maintenance tasks to move partitions, ensuring that there is enough contiguous space to hold a new partition. Rose avoids this problem by allowing tree components to become fragmented. It does this by using Stasis to allocate contiguous regions of pages that are long enough to guarantee good sequential scan performance. Rose always allocates regions of the same length, guaranteeing that Stasis can reuse all freed regions before extending the page file. This can waste nearly an entire region per component, which does not matter in Rose, but could be significant to systems with many small partitions.

Some LSM-tree implementations do not support concurrent insertions, merges and queries. This causes such implementations to block during merges that take $O(n)$ time in the tree size. Rose overlaps insertions and queries with tree merges. Therefore, admission control would provide predictable and uniform latencies. Partitioning can be used to limit the number of tree components. We have argued that allocating two unpartitioned on-disk components is adequate for Rose's target applications.

Reusing existing B-tree implementations as the underlying storage mechanism for LSM-trees has been proposed (45). Many standard B-tree optimizations, such as prefix compression and bulk insertion, would benefit LSM-tree implementations. However, Rose's custom bulk-loaded tree implementation benefits compression. Unlike B-tree compression, Rose's compression does not need to support efficient, in-place updates of tree nodes.

Recent work optimizes B-trees for write intensive workloads by dynamically relocating regions of B-trees during writes (46). This reduces index fragmentation but still relies upon random I/O in the worst case. In contrast, LSM-trees never use disk seeks to service write requests and produce perfectly laid out B-trees.

Online B-tree merging (134) is closely related to LSM-trees' merge process. B-tree merging addresses situations where the contents of a single table index have been split across two physical B-trees that now need to be reconciled. This situation arises, for example, during rebalancing of partitions within a cluster of database machines.

One B-tree merging approach lazily piggybacks merge operations on top of tree access requests. To service an index probe or range scan, the system must read leaf nodes from both B-trees. Rather than simply evicting the pages from cache, this approach merges the portion of the tree that has already been brought into memory.

LSM-trees can service delayed LSM-tree index scans without performing additional I/O. Queries that request table scans wait for the merge processes to make a pass over the index. By combining this idea with lazy merging an LSM-tree implementation could service range scans immediately without significantly increasing the amount of I/O performed by the system.

10.4.2 Row-based database compression

Row-oriented database compression techniques compress each tuple individually and sometimes ignore similarities between adjacent tuples. One such approach compresses low cardinality data by building a table-wide mapping between short identifier codes and longer string values. The mapping table is stored in memory for convenient compression and decompression. Other approaches include NULL suppression, which stores runs of NULL values as a single count and leading zero suppression which stores integers in a variable length format that does not store zeros before the first non-zero digit of each number. Row-oriented compression schemes typically provide efficient random access to tuples, often by explicitly storing tuple offsets at the head of each page.

Another approach is to compress page data using a generic compression algorithm, such as gzip. The primary drawback of this approach is that the size of the compressed page is not known until after compression. Also, general-purpose compression techniques typically do not provide random access within pages and are often more processor intensive than specialized database compression techniques (143).

10.4.3 Column-oriented database compression

Some column compression techniques are based on the observation that sorted columns of data are often easier to compress than sorted tuples. Each column contains a single data type, and sorting decreases the cardinality and range of data stored on each page. This increases the effectiveness of simple, special purpose, compression schemes.

PFOR was introduced as an extension to MonetDB (151), a column-oriented database, along with two other formats. PFOR-DELTA is similar to PFOR, but stores differences between values as deltas. PDICT encodes columns as keys and a dictionary that maps to the original values. We plan to add both these formats to Rose in the future. We chose to implement RLE and PFOR because they provide high compression and decompression bandwidth. Like MonetDB, each Rose table is supported by custom-generated code.

C-Store, another column oriented database, has relational operators that have been optimized to work directly on compressed data (1). For example, when joining two run length encoded columns, it is not necessary to explicitly represent each row during the join. This optimization would be useful in Rose, as its merge processes perform repeated joins over compressed data.

Search engines make use of similar techniques and apply column compression to conserve disk and bus bandwidth. Updates are performed by storing the index in partitions and replacing entire partitions at a time. Partitions are rebuilt offline (22).

A recent paper (62) provides a survey of database compression techniques and characterizes the interaction between compression algorithms, processing power and memory-bus bandwidth. The formats within their classification scheme either split tuples across pages or group information from the same tuple in the same portion of the page.

Rose, which does not split tuples across pages, takes a different approach and stores each column separately within a page. Our column-oriented page layouts incur different types of per-page overhead and have fundamentally different processor cache behaviors and instruction-level parallelism properties than the schemes they consider.

In addition to supporting compression, column databases typically optimize for queries that project away columns during processing. They do this by precomputing the projection and potentially resorting and recompressing the data. This reduces the size of the uncompressed data and can improve compression ratios, reducing the amount of I/O performed by the query. Rose can support these optimizations by computing the projection of the data during replication. This provides Rose replicas optimized for a set of queries.

Unlike column-oriented databases, Rose provides for high throughput, low-latency, in-place updates and tuple lookups comparable to row-oriented storage.

10.4.4 Snapshot consistency

Rose relies upon the correctness of the master database's concurrency control algorithms to provide snapshot consistency to queries. Rose is compatible with most popular approaches to concurrency control in OLTP environments, including two-phase locking, optimistic concurrency control and multiversion concurrency control.

Rose only provides read-only queries. Therefore, its concurrency control algorithms need only address read-write conflicts. Well-understood techniques protect against such conflicts without causing requests to block, deadlock or livelock (16).

10.5 Future work

A number of log-structured index approaches have been recently proposed and fall into a number of different categories. Some, such as partitioned exponential files, use garbage collection algorithms with different (and potentially more difficult to model) performance properties than rolling merges. These techniques use smaller tree components, and, in some cases, perform additional disk probes for each index lookup. Another approach called Hyder does not use B-Tree style nodes, and instead stores a binary tree directly on disk. Its merge process continuously migrates frequently updated nodes into contiguous regions near the "end" of the structure. It assumes the existence of a non-SATA based SSD that provides many more read operations per second than current systems.

Naive LSM-Tree implementations have a number of overheads that would be unacceptable in practice. For instance, without a proper admission control implementation, LSM-Tree write latencies are unbounded. Also, the rolling merge strategy fails to take advantage of common write patterns, such as tree appends.

A new Stasis LSM-Tree is under development. It will address these issues, and include a number of novel optimizations. One of the most important is based upon an algorithm called *snow shoveling* (19) that used to be common when tape drives were popular. Unlike disk based systems, the performance of sorts in a tape based system is highly dependent on the length of the sorted runs that can be produced at runtime. Snow shoveling maintains a heap of data in memory, and tracks the last value written out to the sorted run. Rather than write the heap out in sorted order all at once, it continuously reads new values from the input as memory allows. It always chooses the smallest in-memory value that can be appended to the sorted run. If the input data is in mostly sorted order, this allows it to build runs that are many times larger than RAM. In the random case, this process produces runs that are twice as large as RAM on average (19).

This has a number of immediate implications for LSM-Trees. First, it significantly increases the size of $C0$ for the purposes of the write throughput calculations. Second, for append only workloads, the length of the sorted run is unbounded; therefore, $C1$ will grow indefinitely, and will never require a merge with $C2$. This reduces the amortized insertion cost from $\sqrt{n} log\ n$ to $log\ n$, allowing LSM-Trees to compete directly with B-Tree bulk loads.

Evaluation of this and a number of other LSM-Tree optimizations remains for future work. However, it is interesting to reason about the *safety* of log-structured index optimizations. An unoptimized LSM-Tree is safe in that the amortized cost of each insertion is known *a priori*, allowing admission control to throttle updates, preventing the rolling merges from falling behind. Snow shoveling seems to be a safe optimization; the system could detect when an incoming tuple is past the current merge point, and will therefore be merged to disk before $C0$ fills.

Other optimizations, such as partitioning, complicate the situation. It is unclear what the amortized cost of inserting a tuple into a hot (or cold) partition is; further research will be required before an appropriate admission control algorithm can be developed. For now, LSM-Tree partitioning algorithms are *unsafe*. Categorizing such optimizations, and developing mechanisms that allow them to be applied in practice is a promising direction for further research.

10.6 Conclusion

Compressed LSM-trees are practical on modern hardware. Hardware trends such as increased memory size and sequential disk bandwidth will further improve Rose's performance. In addition to developing new compression formats optimized for LSM-trees, we presented a new approach to database replication that leverages the strengths of LSM-trees by avoiding index probing during updates. We also introduced the idea of using snapshot consistency to provide concurrency control for LSM-trees.

The Rose implementation is a first cut at a building a working LSM-tree. Our prototype's random read performance compares favorably to that of a production-quality B-tree, and we have bounded the increases in Rose's replication throughput that could be obtained without improving compression ratios. By avoiding disk seeks for all operations except random index probes, uncompressed LSM-trees can provide orders of magnitude more write throughput than B-trees. With real-world database compression ratios ranging from 5-20x, Rose can outperform B-tree based database replicas by an additional factor of ten.

However, the experimental results described in this chapter revealed a limitation of Stasis; the buffer manager provided significantly less sequential write throughput than the underlying disk. The problem was due to a bug in Stasis' page writeback policy (Section 12.3) that led to pathological fragmentation. The bug was actually discovered and tracked down by one of Stasis' users while working on another log structured storage system. This reflects both the complexity involved in obtaining good performance from storage implementations, and the power of reusing underlying primitives across many storage approaches. As time progresses, the performance of Stasis' underlying mechanisms has become increasingly well understood; each time a module presents a bottleneck for some class of application, it is improved or replaced with an implementation that performs as expected.

Part III

The Stasis architecture

Chapter 11

Recovery algorithms

11.1 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.

In contrast, Stasis seeks to provide durable, high-performance storage to as wide a range of applications as possible. This chapter covers traditional recovery algorithms, such as ARIES (95), as well as a novel extension to such systems called *segments*. Segments have a number of significant advantages and disadvantages compared pages; therefore, we expect users of segment-based recovery to be interested in hybrid approaches.

Although many systems, such as filesystems and web services, require weaker semantics than relational databases for much of their data, 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. Stasis is based on the idea 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 offs between full ACID and weaker semantics.

Given these broader goals, and roughly twenty years of innovation, this chapter revisits the core of write-ahead logging algorithms, providing correctness arguments for a number of existing and new approaches. These algorithms are the foundation of Stasis.

The new approaches are based upon *segment-based recovery*, and provide more flexibility and higher concurrency, enable distributed solutions, and are simple to implement and reason about.

Segments are based upon revisiting and rejecting 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 (34), and is codified in ARIES (95) and in early books on recovery (17). 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 are covered in Section 11.3. 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 (17; 8). Several mechanisms have been created to make this assumption true with modern disks (36; 124; 145) (Section 11.2.1), 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.

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.

11.2 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 (23).

ARIES and other modern transactional storage algorithms provide Steal/no-Force recovery (58). 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.

11.2.1 Atomic page writes?

Hard disks corrupt data in a number of different ways, each of which must be dealt with by storage algorithms. 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 (36). 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 (103). This can be addressed by disabling write caching, adding an uninterruptable power supply, or by using an operating system that provides synchronous writes.

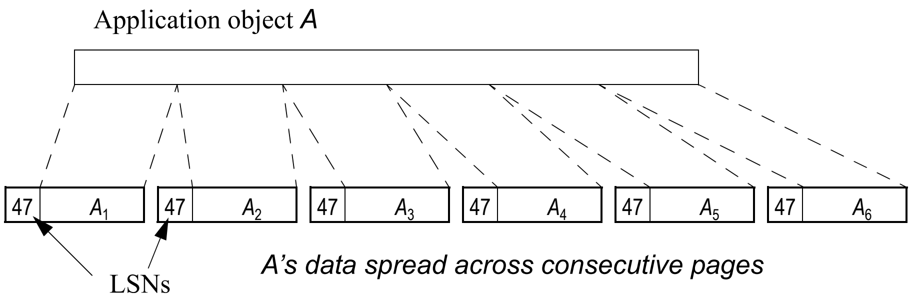


Figure 11.1. Per page LSNs break up large objects.

However, even synchronous writes do not atomically update pages. Two solutions to this problem are *torn page detection* (124), which writes the LSN of the page on each sector and *doublewrite* buffering (145), 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). While this is an improvement over page-based systems, it still does not handle undetected corruption due to hardware errors. Since hardware errors may cause silent data corruption anywhere within a given system, systems that must tolerate such faults typically replicate data across multiple machines.

11.3 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.

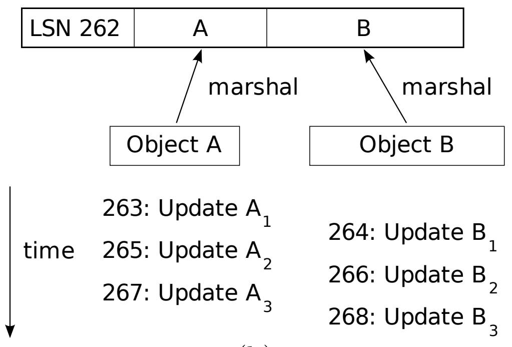
11.3.1 Multi-page objects

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 11.1 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 performance impacts in filesystems (37; 136).

```
pin page
get latch
newLSN = log.write(redo)
update pages
page LSN = newLSN
release latch
unpin page
```

(a)



(b)

Figure 11.2. Record update in ARIES. (a) 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.

11.3.2 Application/buffer interaction

Figure 11.2(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 (54; 61). Other examples include BerkeleyDB (119), systems that treat relational databases as "key-value" storage (145), and systems that provide such primitives across many machines (29; 84).

Figure 11.2(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 (20; 83).

Segment-oriented recovery avoids this and supports write back caching (Section 11.7.2). 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.

11.3.3 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 11.7.3 presents the positive impact this has on high-priority transactions. Before journaling was common, local filesystems supported such reordering. The Echo (85) 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.

11.3.4 Distributed recovery

Page-oriented recovery leads to a tight coupling between the application, the buffer manager and the log manager. Looking again at Figure 11.2, 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.

```
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>;
truncate(min(op_lsn,t_lsn,s_lsn));
```

(c) Truncate log

Figure 11.3. 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.

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 (77) are useful services.

11.3.5 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 during 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.

11.4 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 11.5 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 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

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

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 11.3 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.

11.4.1 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.

11.4.2 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, these entries could be relatively compact, as in Figure 11.4, 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 11.4. In contrast, a conventional approach would simply log the slot number and the new value.

B-Tree concurrency is well-studied (79; 92), 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 can coexist in the same page file, and dynamically share space with each other.

Update N segments

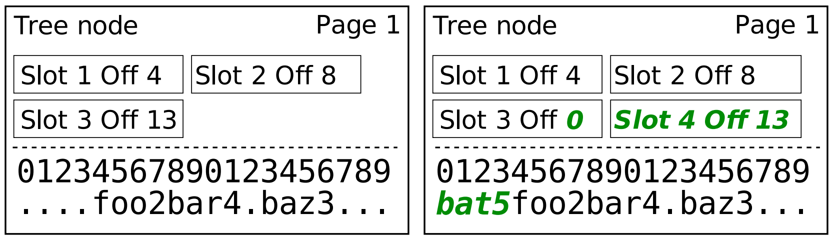


Figure 11.4. An internal tree node, before and after the pair (key="bat", page=5) is inserted.

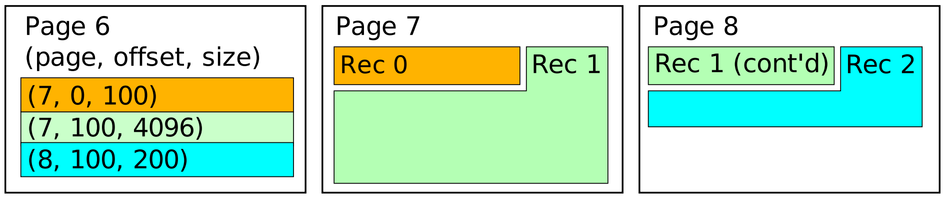


Figure 11.5. Records stored as segments. Colors correspond to (non-contiguous) bytes written by a single redo entry.

```
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 11.5 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 11.7 shows that this can incur significant overhead.

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

11.5 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 (75) 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.

11.5.1 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 (141), 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.

11.5.2 Coherency vs. consistency

We define the set:

$$LSN(O) = \{l :\ O_l = O\} \tag{11.1}$$

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

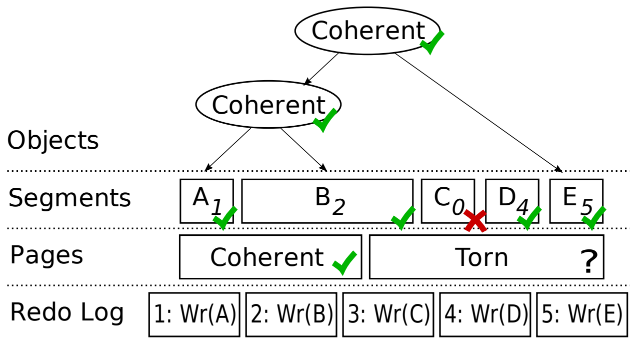


Figure 11.6. State of the system before redo; the data is incoherent (torn). Subscripts denote the most recent log entry to touch an object; Segment C is missing update 3. For the top level object $LSN(O) = \{5\}$. Segment B, the nested object and the coherent page have $LSN(O) = \{2, 3, 4, 5\}$. For the torn page, $LSN(O) = \emptyset$.

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:

$$\exists \text{ segment } s \in O : \forall \text{ LSN } l,\ s \neq s_l \tag{11.2}$$

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 11.6 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):

$$\exists \text{ LSN } l : \forall \text{ object } o \in O, l \in LSN(o) \tag{11.3}$$

Lemma 11.5.1 *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 11.5.6 describes a runtime latching and logging protocol that guarantees undo's logical operations only encounter consistent objects.

11.5.3 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.

11.5.4 Write-ahead and checkpointing

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

$$\forall \text{ segment } s : \exists\, l \in LSN(s_t^{stable}) : l \leq log_t^{stable} \tag{11.4}$$

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

$$\forall \text{ segment } s, \exists\, l \in LSN(s_t^{stable}) : l \geq log_t^{trunc} \tag{11.5}$$

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 11.4 or 11.5:
$\exists \text{ redo } e : e.lsn \in \{l : log_t^{trunc} \leq l \leq log_t^{stable}\}$:

$$e.lsn \in LSN(e(\top)) \tag{11.6}$$

Where $e(\top)$ is the result of applying e to a corrupt segment. This will be needed for hybrid recovery (Section 11.6.2).

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

11.5.5 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:

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

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:

$$\forall \textsf{ segment } s, \exists\, l \in LSN(s_t^{current}) : l \geq log\_cursor_t(s) \tag{11.7}$$

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,

$$s_{l-1} = s_l \vee (\exists\, e : e.lsn = l \wedge e.segment = s) \tag{11.8}$$

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 11.7). The hybrid approach is more complex and relies on allocation policies (Section 11.6.2).

11.5.5.1 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.

11.5.5.2 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.

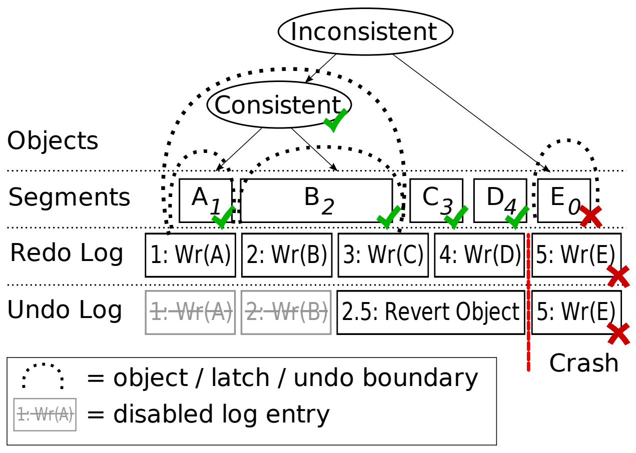


Figure 11.7. State of the system before undo; the data is coherent, but inconsistent. At run-time, updates hold each latch while manipulating the corresponding object, and release the latch when they log the undo. This ensures that undo entries never encounter inconsistent objects.

11.5.5.3 Proof of redo's correctness

Theorem 11.5.2 *At the end of redo, the database is coherent.*

Proof *From the definition of coherency (Equation 11.3), 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 11.4 and 11.7 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.

11.5.6 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 11.7).

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.

Recall that the definition of *coherent* (Equation 11.3) 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 (47).

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. More permissive approaches (79; 92) 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 $e, f : e.tid \neq f.tid,$

$$o = e.object = f.object \Rightarrow e(f(o)) = f(e(o)) \tag{11.9}$$

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 11.5) 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 11.9 are due to two transactions directly invoking two non-commutative operations. This is a special case of *write-write conflicts* from the concurrency control literature; in the absence of such conflicts, Equation 11.9 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 11.9.

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.

[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	Reallocate	Page	Segment	Other xact	Same
1	Y		Y	Y	Y	Y
2		Y	Y	Y		Y
3		XOR	Y			Y
4	Never		Y	Y		

Figure 11.8. Allocation strategies. The first two columns describe the logging strategy—whether various operations log the old contents of the space. The next two columns describe the approaches' compatibility with pages and segments, and the final columns describe compatibility with schemes that reuse space before the freeing transaction commits.

11.6 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 11.8 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* (78) 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.

11.6.1 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 (93). 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 (93).

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 (82).

11.6.2 Correctness of hybrid redo

Here we prove Theorem 11.5.2 (redo's correctness) for hybrid ARIES and segment-based recovery. The hybrid allocator zeros out pages as they switch between page- 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:

$$e.lsn\_free \iff lsn\_free(e.segment_{e.lsn}) \tag{11.10}$$

Theorem 11.6.1 *Hybrid redo leaves the database in a coherent state*

Proof *Equations 11.4 and 11.5 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 11.5.5) this will maintain the invariant in Equation 11.7 until it completes.

In the second case, the assumption is incorrect. By Equation 11.10, the stable version of the page must have a different 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 11.4, 11.5, and 11.8) 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 11.6). At this point, the page format matches the current log entry, reducing this to the first case.

11.7 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.

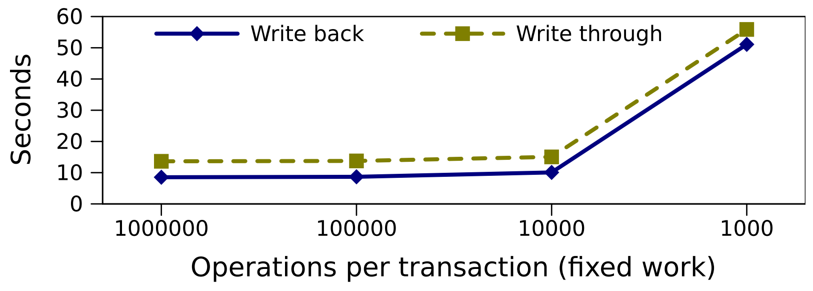


Figure 11.9. Time taken to transactionally update 10,000,000 `int` values. Write back reduces CPU overhead.

11.7.1 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.

11.7.2 Write caching

Read caching is a fairly common approach, both in local and distributed (40) 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. 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.

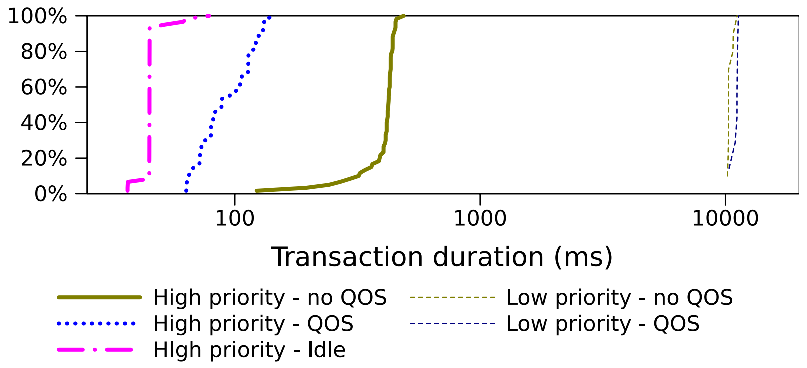


Figure 11.10. CDF of transaction completion times with and without log reordering.

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 cache eviction. This increases CPU costs substantially.

The effects of extra buffer management overhead are noticeable even in the single-threaded case; Figure 11.9 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.

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

Figure 11.10 plots the cumulative distribution function of the transactions' response times.

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 11.11. 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.

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.

11.7.4 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. Our simple network model ignored issues such as network topologies, and did not limit available bandwidth between components. 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 11.11), 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.

11.8 Other request reordering and distributed logging schemes

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 (85). Its *write-behind* queues provide rich write-reordering semantics and are a non-durable version of our reorderable write-ahead logs.

FeatherStitch (42) 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 (40) 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 (41), which partition data across independent index nodes, and Boxwood (84), which distributes indexes across clusters, are two extreme points in the scope of possible designs. A third approach, Sinfonia (4), has nodes expose a linear address space, then performs *minitransactions*; essentially atomic bundles of test and set operations against these nodes. In contrast, page write-back 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 (20). Application specific systems also exist, and handle atomicity in the face of unreliable clients (99).

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 (83). 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.

11.9 Summary

This chapter covered traditional recovery algorithms, and segment-based recovery. Segments break the coupling between page boundaries and the granularity of application requests by removing LSNs from pages. This brings request reordering and reduced communication costs to concurrent, Steal/no-Force database recovery algorithms. This chapter 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 the 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.

Stasis does not yet include practical support for segment-based storage. However, we believe that the mechanisms described in the following chapters provide a strong foundation for segment-based approaches; in particular, Stasis' decoupling of recovery mechanisms from the underlying data format simplify the changes that must be made, and the similarities between ARIES and segment-based approaches suggest that the changes to mechanisms such as recovery, the dirty page table, and logging will be minor.

Chapter 12

Buffer Management

The buffer manager provides a conceptually simple API; it allows higher-level code to pin and unpin pages. This is all that is required for conventional Steal/no-Force recovery. However, for the buffer manager to be truly general-purpose, it must support a number of other features.

First, each thread must be able to pin arbitrary numbers of pages (up to the size of RAM). This is needed for no-Steal transactions. Because the buffer manager does not understand the structure of the data it stores, it must not impose latch orderings or other constraints on the protocols used to pin pages. This allows higher-level code, such as tree structures, to perform pins in an order natural to the structure of the underlying data.

Second, the buffer manager needs to allow applications to force write pages to disk. Two force modes are required. The first is simply a performance hint, and does not need to succeed. The second mode is used for correctness, typically to write pages to disk before committing a Force mode transaction or truncating the log.

Ignoring hardware failures, the only reason a dirty page may fail to be written back is because that page is currently pinned for update. If the buffer manager detects this, it returns an EBUSY error code. It is up to higher-level code to ensure that no pages scheduled for force writes are pinned at transaction commit.

Finally, the buffer manager needs to support callbacks that are triggered each time it reads and writes pages to disk, and when it evicts a page from memory. This allows certain operations, such as compression and integrity checks, to be amortized across many in-memory operations. Callbacks are associated with *page types*, which are either recorded in the page headers or are passed alongside requests to pin pages.

The remainder of this section describes the modules that provide this functionality. Portions of the buffer manager implementation, such as hardware interfaces, write-back policies, and page eviction policies, are likely to change frequently, while other aspects, such as state for recovery and concurrency mechanisms, change infrequently. Therefore, this functionality is factored into separate sub-modules.

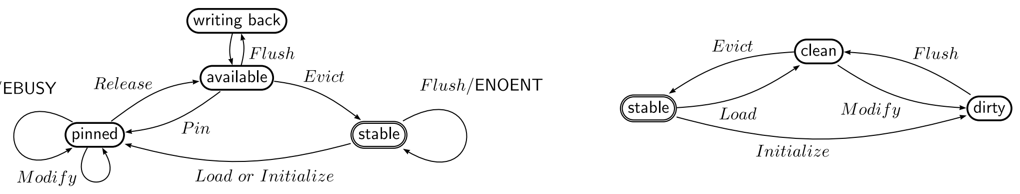


Figure 12.1. Two coupled state machines modeling the status of Stasis pages.

12.1 Buffer management state machines

Figure 12.1 describes the possible states of Stasis pages. Pages start in the stable state, and return to it when they are evicted from memory. If a transition is present in both machines, then both machines must be in a state that accepts the transition for either to accept it. For instance, a page must be clean and available before it can be "Evicted." The "/" indicates error codes; "Flush" only moves a page from clean to dirty if the page is available.

These state machines are exposed though various portions of the buffer manager API. "Pin/Load" and "Release" correspond to the calls that provide access to pages. Whether a page is clean or dirty is stored in the page's dirty bit. Finally, page formats may register callbacks that are executed on each "Flush," "Load," and "Evict." Each page format also provides methods to "Modify" and "Initialize" pages.

Although this model has served Stasis well, its treatment of read-only data could be improved slightly. Stasis does not distinguish between pages that are pinned for reads and those that are pinned for writes. A "read-only pin" state would allow pages to be written back (but not evicted) while being accessed by the application . However, this would complicate the buffer manager API somewhat, as higher-level code would have to decide whether it may want to write to a page, or whether it should pin for read-only access.

12.2 Page types: callbacks and cached state

The buffer manager maintains a few additional pieces of state per page. It must keep track of each page's type so that it can invoke appropriate callback functions. It also must keep track of the LSNs of the first and last operations that modified the page. The first LSN is needed so that the log can be safely truncated. The last is needed in order to maintain the write-ahead invariant; before writing back each page, the buffer manager checks to see if the log should be force-written to disk.

In the common case, page types and the last LSN are stored in the first few bytes of each page; the buffer manager examines these bytes and sets its internal state appropriately. There are two exceptions to this rule. First, a page might be "uninitialized;" its page ID may have just been returned from the allocator. The first few bytes of such pages may contain arbitrary values; so an appropriate type must be provided by the caller. Second, a page may be formatted for segment-based recovery (Section 11.4), and have no header at all. Again, the caller must pass in appropriate type information for such pages. LSNs for pages with headers are computed using a callback function that simply reads the LSN from the page header, while LSNs for segments are estimated by a different callback function.

Introducing the idea of page types creates a new complication. When a page changes type (due to allocation or being reinitialized), it must cleanly transition from one type to the next. This is handled by invoking the "Evict" callback on the page, then passing the page into the new type's initializer. In addition to the LSNs and page types, the buffer manager maintains a single `void*` pointer and latch per in-memory page. The latch protects any shared state associated with the page, while the pointer provides access to page-type specific information. This allows page implementations to cache information that is expensive to compute. For instance, compression formats may keep buffers of uncompressed data, and index headers may contain cursors pointing to recently accessed data. Section 9.7.3 explains how to make effective use of these mechanisms in more detail.

Because this information is associated with an in-memory page, the "Evict" callback must be prepared to free it whenever the page is clean; similarly the "Flush" callback must be able to write any cached updates to the page whenever the page is not pinned.

According to the state machines, a page must be clean before it is evicted. If the page is dirty, the buffer manager invokes the "Flush" callback, then immediately invokes "Evict" without actually flushing the page to disk. This avoids an unnecessary disk write without complicating the "Evict" callback. However, it does mean that "Flush" may be invoked even when the buffer manager has no intention to flush the page.

12.3 Write-back

Stasis' buffer manager is split into a number of different components, allowing portions of the buffer manager to be replaced without modifying the rest of the system.

Two such components are the page write-back and eviction policies. Write-back decides when, and in what order dirty pages should be written to disk, while page eviction decides which clean pages should be removed from memory in order to make room for newly requested pages.

Although the purpose of these two modules is similar, write-back must be careful to schedule I/O in a way that provides good disk throughput, while eviction should avoid getting rid of pages that will be accessed again in the near future. Stasis' current policies are fairly simple. Write-back sorts the dirty pages in file order, then iterates over the list, writing back each page in turn. The write-backs are asynchronous. If they were synchronous, only one page would be available to the operating system at a time, so only one device in the storage hierarchy could move data at once. Worse, after writing a page, the drive wouldn't have any additional data to write. In order to write the next page, it would have to wait for an entire platter revolution.

The current writes are only asynchronous to the drive platter; pages are still passed to the kernel in a synchronous fashion. Also, Stasis only makes use of a single write-back thread. One might imagine that more complex schemes (such as multiple write-back threads) would provide greater concurrency and throughput. However, when current versions of Linux are presented with out-of-order writes to newly allocated portions of a file, they create a sparse file, leading to pathological filesystem fragmentation. In such cases, the throughput of the first write is reasonable, but subsequent sequential reads and writes incur a seek per page. The simple approach of passing pages one at a time, in order, leads to good, predictable performance. Since the kernel returns control to Stasis immediately after the page is copied to kernel RAM, the operating system and drive write caches will have many outstanding page writes to pipeline before the first page is written to disk.

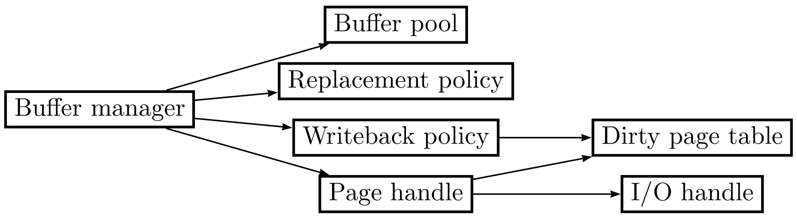


Figure 12.2. Buffer manager subcomponents.

Two additional factors complicate the current scheme. First, Stasis should not start writing pages back until there are enough to pipeline data, but not so late that the disk becomes idle while the CPU is busy dirtying pages (or the CPU goes idle waiting on disk writes). Second, once Stasis starts copying data to kernel space, it should avoid stalling due to latches held by processes that are dirtying additional pages, or due to latches held by the write-back thread. Currently, it chooses a batch of pages to write-back all at once, copies the list into local storage, then writes back each page in the list in turn, skipping latched pages.

Although this provides reasonable sequential and random write throughput for most workloads on current hardware, changing hardware characteristics and update patterns have obsoleted earlier Stasis write-back policies; it would not be surprising if other policies are more appropriate in the future.

12.4 Page eviction

Like write-back policies, Stasis' page eviction policy is implemented as a standalone module. Stasis currently uses LRU replacement. However, more complicated schemes such as LRU-2S would likely perform better (86), and are drop-in replacements.

A number of other eviction policies make use of extra information about the workload to make better decisions regarding page eviction. Such eviction policies allow higher levels to reserve (and reason about) free ranges of memory. Because access patterns are often known to the application ahead of time, they also allow certain page ranges to be managed using different policies, such as evicting pages immediately, or using MRU for data that does not fit in RAM, but that will be repeatedly scanned (135). Stasis currently does not support such schemes, and existing approaches hard-code assumptions about higher-level primitives. Formulating a general-purpose API for page eviction hints is left for future work.

12.5 Implementation details

Stasis' default buffer manager is factored into a number of sub-modules (Figure 12.2). The buffer manager interface provides the public methods that are used by higher-level code. It is possible to implement a from-scratch buffer manager without using any of the other modules (except for the dirty page table, which is also used by the log manager). However, the current design decouples a number of otherwise unrelated mechanisms. The current architecture is the third rewrite of Stasis' buffer manager. Unlike its predecessors, it has proven to be correct, performant, and maintainable.

The default buffer manager is backed by an in-memory hash. The complexity in this module comes from concurrency; to obtain good performance, buffer managers must be able to service multiple concurrent requests even while requests to read and write pages block on disk activity.

The buffer pool manages memory allocation for the space that backs in-memory pages. It is fairly straightforward, except that pages are aligned in memory according to the filesystem sector size (512 bytes). This is required by some of the I/O interfaces provided by the kernel.

The page handle enforces the write-ahead invariant and maps between page IDs and disk offsets. It enforces write-ahead by consulting the dirty page table; if necessary, it force writes the tail of the log before writing a page to disk. Page handles communicate with storage via Stasis-specific file handles. This allows a single buffer manager implementation to support many types of storage hardware and multiple kernel APIs.

The write-back and replacement policies were covered above. The write-back policy consults the dirty page table when deciding which pages should be written back.

12.6 Exotic buffer managers

Stasis' default buffer manager is designed to provide good performance across a wide variety of workloads. There are at least two other approaches that may perform better in certain circumstances. The first, `mmap`, relies on special hardware and operating system features that are not widely available. The second, *log-structured* buffer management, guarantees that writes are sequential. However, it must garbage collect written back pages, at the cost of a significant amount of sequential I/O.

12.6.1 Buffer managers based on `mmap`

Operating systems maintain a cache of disk pages, and provide some degree of control over when pages are read from and written to disk. In principle, these mechanisms could be used in lieu of a buffer manager. This has a number of advantages; in particular, pages would be laid our sequentially in virtual memory, and user space code would be significantly simpler. The main disadvantage is that such schemes can only access databases as large as virtual memory, which is currently a bit under 48 bits worth of address space.

12.6.1.1 `mmap` still provides poor semantics

Historically, operating systems have provided primitives that are inappropriate for for database implementations (126). Although the situation has largely been addressed by standard file access methods, threading models and other modern operating system features, it is still difficult to use `mmap` for buffer management.

The `mmap` system call seems to provide a reasonable starting point; it maps a region of disk to a region of virtual memory, automatically reads pages, and propagates changes to the disk. Two related primitives, `msync` and `mlock`, provide functionality analogous to force and pin, respectively. A fourth call, `madvise` passes hints to the kernel's write-back and page replacement policies.

Unfortunately, for historic reasons, these primitives were largely used for two purposes, and have not been extended to support other applications (91). The first purpose, secure handling of authentication tokens, uses `mlock` to prevent encryption keys from being written to the swap file, and relies on the fact that the `mlock`'ed memory is on the heap. The second, real-time applications, ensures that pages remain resident, but does not ensure they will not be written back to disk.

In fact, `mlock` must allow such pages to reach disk. Linux provides a *coherent* memory architecture; once a byte is written to an `mmap`'ed page, it becomes available to other applications running on the system. Therefore, two applications will never observe out-of-sync versions of the same page. Although elegant, this prevents `mlock` from actually pinning pages in memory.

To see why, consider an update to an `mlock`'ed page that has been observed by a second process that read the file from the filesystem. At this point, the filesystem cache must contain the update (and perhaps updates to the same file from other processes). If the process that `mlock`'ed the page were to crash, then Linux would need to revert the pages that process updated to their original values (which may have never been written to disk). Correctly reverting these updates would force Linux to track provenance and keep multiple versions of dirty pages.

Therefore, when combined with `mmap`, `mlock` constrains Linux's page eviction, but not write-back. Note that this is not the case for memory is backed by the swap file; `mlock`'ing such memory constrains both page eviction and write-back.

A second set of primitives, *shared* and *private* mappings, also seem promising at first glance. Updates to a shared mapping are propagated to other processes, while updates to private mappings are local to a single process. Unfortunately, a given memory mapping cannot be switched between shared and private mode. Otherwise, one could pin a page by marking it private, and release the page by marking it shared.

As a last resort, one might attempt to perform zero-copy I/O from a private mapping back to the shared disk cache. In principle, such an operation would be inexpensive; the private copy was created using copy on write, so Linux could simply drop the original version of the page and replace it with the new copy. However, this would require special purpose kernel code. Currently, if the Linux kernel detects such a request, it responds with an I/O error.

Note that such a mechanism would reduce (but not eliminate) the problem of keeping two in-memory copies of the buffer manager's contents, without resorting to `O_DIRECT`. `O_DIRECT` is a flag that can be passed into file open requests. It disables the operating system cache for the opened file, essentially breaking kernel-level optimizations such as I/O scheduling.

12.6.1.2 Reusing kernel-level write-back and page eviction

The next piece of the puzzle is `madvise`, which provides an appropriate API for buffer manager code; it passes hints to the kernel, allowing programs to indicate that data will be read sequentially or randomly, and whether data should be kept in cache, or immediately dropped. Relational databases typically implement such mechanisms internally.

However, databases also keep statistics regarding the current utilization of their buffer pool, and the status of the cached pages. These statistics are used for query optimization and admission control (135). Analogous operating system mechanisms are currently unavailable. Although not as complete as approaches used by relational databases, `madvise` still provides richer semantics than the current version of Stasis; page replacement policies are an area where current operating systems shine.

Finally, in order to implement a buffer manager, one must deal with write-back, both for performance and for correctness. The hints provided by `madvise` partially address performance, but the correctness issues are more complex.

The issue is that write-ahead logging's dirty page table contains significantly more information than is available from the OS; namely the timestamp of the first and last modification of each in-memory version of a page. Without additional operating system or user-space mechanisms, such data must be approximated.

One approach is to periodically `msync` the entire `mmap`'ed region to disk, and to remember when the last such sync began. Although this works, it may result in I/O patterns that are significantly more expensive than necessary. Typical database write-back policies prioritize disk writes according to update order to reduce the number of pages written before each log truncation.

It also is unclear what semantics an `msync` of a pinned page should have; in order to guarantee that all pages are written back by an `msync` of the entire page file, the system must ensure that no pages are currently pinned. Again, this leads to the sorts of user space bookkeeping that we hoped to avoid by using `mmap`.

12.6.2 Log structured buffer management and snapshot recovery

Traditional write-ahead logging systems assume that data is written to disk a page at a time, and that pages are updated in place. Systems based on shadow copies are similar. Although the majority of data is written to free space, rather than in place, at commit, they must atomically set a pointer stored in a page in the buffer manager.

Snapshot recovery schemes behave differently, and rely upon operations that atomically replace the entire page file at once. Such schemes are particularly attractive for systems with high write throughput and relatively small databases (113). Because the snapshot atomically updates the disk, such systems avoid the need for physical redo and nested top actions; in order to commit a transaction, these systems force logical redo entries to log. For some workloads (such as SQL update statements that compute updates to many records), these redos are significantly smaller than the corresponding physical log entries.

Alternatively, if losing recent updates on crash is acceptable, snapshot-based systems can avoid logging entirely. This provides recovery with predictable overhead; for these reasons, soft-realtime systems such as video game servers often use snapshot recovery (113).

12.6.2.1 Write-back strategies

However, these reductions in log bandwidth come at a significant cost; scaling snapshot recovery to large amounts of data can be difficult. If the database is larger than RAM, then the next snapshot cannot easily be written without performing a merge of on-disk and in-memory state. The discussion of log structured merge trees (LSM-Trees) in Section 10.1 explains how arbitrarily large tree structures can be updated using purely sequential I/O. The basic idea is to repeatedly merge the in-memory data with a "small" on-disk component, then occasionally merge the small component with a larger on-disk component.

The solutions for snapshotted buffer managers are analogous, except that, because data is written with page granularity, there is never any reason to read existing data during merges. This is not the case with LSM-Trees, as overwritten and current tuples may coexist on the same disk pages.

The relative performance of the two approaches depends on update patterns and hardware characteristics. If updates are sparse, then most of the bytes written by page-level merges will be unchanged. Tree-based systems only store the changed values, leading to a significantly more compact representation of the updates. However, since tree-based systems must merge in-memory and on disk state, their merge processes must touch a significantly larger fraction of the database than page-based approaches.

Broadly speaking, page-based approaches may either lay out data sequentially, increasing the cost of merges, or they may lay out data randomly, increasing the cost of scans, and (perhaps) of locating a page on disk. The fact that data may be stored on a hard disk or on an SSD further complicates matters.

Like LSM-Trees, buffer managers that update disk using repeated merges can atomically snapshot database state. However, coherent snapshots of memory do not guarantee the system is recoverable on their own (Section 11.5.2). With page-based merges (which have less information about transactions and high-level atomicity guarantees), one must ensure that no updates are in progress while the snapshot is taken. In general, this means that there may be no outstanding transactions; all updates must be quiesced. Techniques such as copy on write allow updates to proceed during the actual disk write back.

Even with such optimizations, the cost (and complexity) of periodically quiescing the system may be significant; it is proportional to the product of the number of concurrent transactions in the system and the length of the longest running transaction.

In contrast, if updates are tagged with transaction timestamps that correspond to a serialized schedule, then it is trivial to produce an atomic snapshot of a tree-based system.

There are a number of tradeoffs between this scheme, which performs merges below the buffer management layer, and schemes such as LSM-trees, which merge above the buffer management layer. A careful examination of these tradeoffs is left for future work.

12.6.2.2 Limitations of snapshotting

Given that snapshot-based schemes use significantly less log bandwidth than full redo/undo logging, one may wonder why ARIES-style recovery has become the default approach and snapshot based schemes are considered an “exotic” approach. Although there are a number of reasons, the three most important are:

Not quite Steal/no-Force Although write-back can be largely decoupled from snapshot boundaries, the fact that pages are updated out-of-place, and the fact that the same page may be stolen many times means that potentially complex mechanisms are needed to ensure that the correct version of each page is read back at runtime and at recovery. Also, these systems must force each page at each snapshot; this is analogous to forcing pages during ARIES-style log truncation, but interacts with attempts to work around the next limitation.

Subtle incompatibilities with two-phase commit In theory, snapshot-based systems support two-phase commit. In practice, other systems participating in the two-phase commit may communicate with the snapshot based system after the logical redo (application request) is generated. If this happens, any information provided by the other systems will not be present in the log after recovery, making it difficult for snapshot based systems to correctly implement "prepare." Simply replaying the requests to the external systems at recovery is inadequate, unless the requests are idempotent and deterministic (regardless of whether or not the other system has committed the transaction).

Need to quiesce the system This was mentioned above, but bears repeating. The system must be prepared to quiesce while the longest running transaction completes. In highly concurrent systems, each snapshot may waste significantly more processor time and I/O operations than are used by the longest transactions.

Other significant issues with snapshot-based schemes include lack of support for media recovery and the possibility of breaking recovery by starving the snapshot/log truncation thread. With ARIES, log truncation immediately frees disk space. Snapshot schemes must write back an entire snapshot before truncating the log. This prevents snapshot based schemes from reclaiming space once the disk is full.

With ARIES, it is sufficient to bound the length and number of outstanding transactions (including space required for aborts) and size of the database. Snapshot schemes must also enforce hard limits on the amount of disk required by the current and in-process snapshot, log entries of committed transactions in the in-process snapshot, and any unreclaimed space taken up by prior snapshots. From an engineering perspective, this means that a number of modules that are simply optimizations in ARIES are crucial to the correctness of snapshot-based schemes.

12.6.3 A case for multiple page sizes

When applications manipulate more data than can fit in RAM, the system must service buffer management requests by accessing disk. Depending on access patterns, and characteristics of the underlying hardware, performance may heavily depend upon the size of pages read from and written back to disk.

For instance, the internal nodes of a very large B-Tree index may not fit in RAM. If we think of each B-Tree node as though it encodes some subset of a binary tree, and we know how many entries will fit on a page, then we can compute the number of levels of the binary tree that can be traversed by reading a single page. As page sizes increase, more levels are traversed per disk read, but each read becomes more expensive and brings increasing amounts of useless information into the buffer cache. Derivations of optimal page sizes, given hardware characteristics such as seek times and read bandwidth are well-understood (50).

However, that analysis only provides guidance regarding the size of internal nodes. If we assume that hot and cold data are randomly interspersed throughout the tree, then increasing the page size simply increases the fraction of the buffer pool that will be polluted with cold data that happens to on the same page as hot data. Therefore, for random access patterns, the smaller the leaf nodes, the better (assuming we ignore internal fragmentation).

Models of access patterns, and subsequent system tuning can be quite complex, but that is not the point of this section. The important point here is that it is possible that some pages in the system should be one size, while others should be some other size.

Systems with multiple page sizes are likely to increase in importance for a few reasons. First, although improvements in sequential hard disk bandwidth have outpaced improvements to seek times, decreasing the incremental cost of reading and writing large pages, in some sense, flash-based solid state disks turn the clock back a bit. Flash provides significantly more I/O operations per second than hard disks, which increases the relative importance of the read bandwidth saved by decreasing the size of leaf nodes.

Second, many workloads fit entirely in RAM; in such systems, log writes and periodic write-back of pages dominate I/O costs, while the cost of pinning pages dominates CPU costs. Since solid state drives use log-structured disk layouts; in principle, the cost of each write is proportional to the amount of data written. However, in practice, the cost of garbage collection determines the write throughput of the device, and small writes increase garbage collection costs. If the situation improves, then it is likely that optimizations that decrease the amount of data written back without changing logical layouts will increase in importance.

Chapter 13

Logging

Fundamentally, Stasis manages two types of objects at runtime: pages (or segments), and transactions. The previous chapter explained Stasis' buffer manager, and how page states evolve over time. This chapter describes the logging system, which manages transactions at runtime and during recovery. It is useful to think of operations to the buffer manager as though they are partitioned according to page, and operations that affect the log as though they were partitioned according to transaction. Thus, this chapter begins with a discussion of Stasis' transaction table; the log's analogue to the dirty page table.

13.1 Transaction table

The transaction table is one of a few pieces of global state maintained at runtime. It tracks two pieces of information for each transaction: the LSN of the first log entry generated by the transaction and the LSN of the transaction's most recent entry. The transaction table is also responsible for assigning transaction IDs. Finally, it provides an API that allows higher-level modules to associate state with transactions, and to register callbacks that will be invoked immediately before the transaction commits. This is particularly useful for modules such as lock managers, allocation policies (Section 11.6), and two-phase commit (Section 7.12).

A number of different approaches to transaction ID assignment are possible, including *sequential*, where transaction IDs are not reused, and *thread-local*, where each thread maintains a pool of transaction IDs that has reserved. The threads then choose values from the pool, reusing them as necessary. This avoids synchronization overheads between transactions running in separate threads. It also increases cache locality of any transaction specific data structures.

13.2 Log API

Stasis' log is conceptually simple. It supports truncation, forward and reverse iteration, and a number of force-write modes. Forward iterators produce each entry in chronological order, while reverse iterators are associated with a transaction.

In order to ensure forward progress during recovery, each time an entry is undone, a compensating entry is appended to the end of the log. Upon encountering a compensating entry, reverse iterators skip back to the entry that the transaction produced before the compensated one.

Like the buffer manager and transaction table, Stasis' log is performance critical. Each request to update the buffer manager generates a log entry; serializing these requests can have a major impact on request concurrency. The current log API takes one log entry at a time, appends it to an in-memory buffer, then returns the LSN (offset into the log) of the entry. Segment-based recovery relaxes this, allowing the LSN to be returned asynchronously and with coarser granularity.

13.2.1 Three LSNs

Ignoring the actual contents of log entries, the log's state can be summarized by three monotonically increasing LSNs. The first LSN, *trunc_lsn* records the beginning of the first complete entry in the log. The second LSN, *stable_lsn* records the LSN of the last entry that has made it to stable storage. The third LSN, *commit_lsn* is the LSN of the most recent committed entry.

13.3 Force writes

Stasis' log maintains the current *stable_lsn* to ensure that the system is recoverable, and the current *commit_lsn* to ensure that important updates are not lost. By default, these two numbers are the same. By varying their semantics, Stasis applications can trade off between performance and durability without worrying about leaving Stasis in an unrecoverable state at crash.

To this end, Stasis provides a *softcommit* mode that allows transactions to "commit" before they reach disk; when transactions "force" the log to disk, the logger ignores the request. However, it still must force write the log before the buffer pool steals pages; otherwise, Stasis would violate the write-ahead invariant and leave the system in an unrecoverable state. Therefore, before writing pages back to disk, the buffer pool forces the log to disk, if necessary. Such requests have a flag set so that they can be distinguished from requests to force write at transaction commit.

13.4 Log entry format

Stasis log entries are based upon an extensible format. Each entry (regardless of type) consists of three values: *previous LSN*, *transaction ID*, and *entry type*.[1] If the previous LSN is `INVALID_LSN`, then the entry is at the beginning of a transaction. The entry type can be one of a handful of values, such as "begin," "commit," "abort," "update," and "internal."

13.4.1 "Update" entries

From higher-level code's point of view, update entries are the most interesting type. Each update entry contains an op code, and a bit of extra information that constitutes arguments to the oper-

[1] The size of the log entry is stored in a log file-format dependent way, and is also implicitly encoded in the entry. Otherwise, it would be difficult to determine how much memory to allocate when reading the entries from disk and when manipulating them in memory.

ation, including information needed by recovery. One such argument is the offset of the page that the update modifies.

Physical operations set the page id, while logical operations store `INVALID_PAGE` in this field. A byte array is appended to the end of each update entry; this contains the operation-specific data. For physiological operations, it typically consists of the before and/or after image of the item that was modified.

13.4.2 "Internal" entries

It quickly became apparent that real log file formats need to store additional bookkeeping data, such as CRCs of regions of the file, information about truncation, and so on. Although some of this information can live in a fixed-length area at the beginning of the file, this data is often variable length, or should be written at the same time as higher-level records. (For instance, a CRC is written after each request to force write the log to disk.) One could imagine implementing a log file format that would filter out such data automatically, but this would complicate log implementations substantially.

Stasis' log file implementations only support a handful of simple operations, such as "read entry at file offset," "append entry," "truncate log to offset," and "force write log to offset." Higher-level code provides forward and reverse iterators over this API, hiding transaction and recovery details from the implementation of the on-disk format. Allowing the log file format to add "junk" between entries breaks the abstraction.

Internal entries address these issues without hard-coding details of the underlying log representation in higher-level code, and without complicating the log implementation. The log implementation may append "internal" entries to the log at any time. When higher-level code encounters such an entry, it simply ignores it.

13.5 External write-ahead

Stasis has been deployed in a number of environments that provide their own write-ahead primitives. The first such environment is *Valor*, a Linux kernel extension that arranges for writes to one partition to be force written before writes to the others (122). By running Stasis in softcommit mode, one can completely avoid force writing the log to disk. If the log is stored on the high-priority partition, then forcing a buffer manager page to disk will first force any outstanding log writes. Therefore, flushing the log to the kernel page cache is sufficient.

Furthermore, if commits must happen before some other filesystem updates take place, then placing the filesystem updates on the low-priority device and performing the operations after Tcommit() returns will ensure that Stasis' commits happen before they side-effect the filesystem. We used this approach as part of a high write throughput filesystem provenance database (123).

With such an approach, the only force writes are performed in order to allow the log to be truncated; log writes that maintain write-ahead are naturally ordered before writes to the page file.

Chapter 14

Cross-component interactions

The previous two chapters described Stasis' buffer manager and logging infrastructure. In isolation, these two components are reasonably straightforward. This chapter describes how the systems interact in order to provide correct recovery semantics, bridging the gap between the theoretical treatment in Chapter 11 and the practical issues confronted by recovery implementations.

14.1 Tupdate(): Performing an operation

The `Tupdate()` function performs most updates to Stasis' page file. It is responsible for pinning pages, invoking redo callbacks and writing entries to the log. In turn, these operations modify the dirty page table and transaction table. `Tupdate()` hides this complexity behind a narrow API. It takes as arguments a transaction id, a page id, an op code, and a byte array that contains the argument that will be passed to the operation's physical redo and undo methods.

Unless segment-based recovery or other concurrency optimizations are in use (Section 11.7), each call to `Tupdate()` is atomic. Since each physical operation manipulates a single page, it is sufficient to guarantee that updates to each page are applied sequentially and in log order. This is easily guaranteed by keeping the page in question pinned and latched as the log entry is generated and appended to the log buffer.

14.2 Concurrency: Logical undo and nested top actions

`Tupdate()` handles the generation of physical redo and undo entries. In order to support concurrent transactions, Stasis also provides logical undo and nested top actions. Section 11.5.6 describes how these mechanisms work together to implement multi-level recovery. From Stasis' perspective, the only important detail is that each time a high-level "logical" action is taken (such as inserting an item into an index), the lower-level steps that perform the action must be wrapped in a nested top action that records a logical undo in the log.

Determining which operations are "high level" and formulating appropriate latching policies is of no concern here. Instead, they are covered in Section 9.3, which provides guidance for implementers of concurrent indexes and other durable data structures.

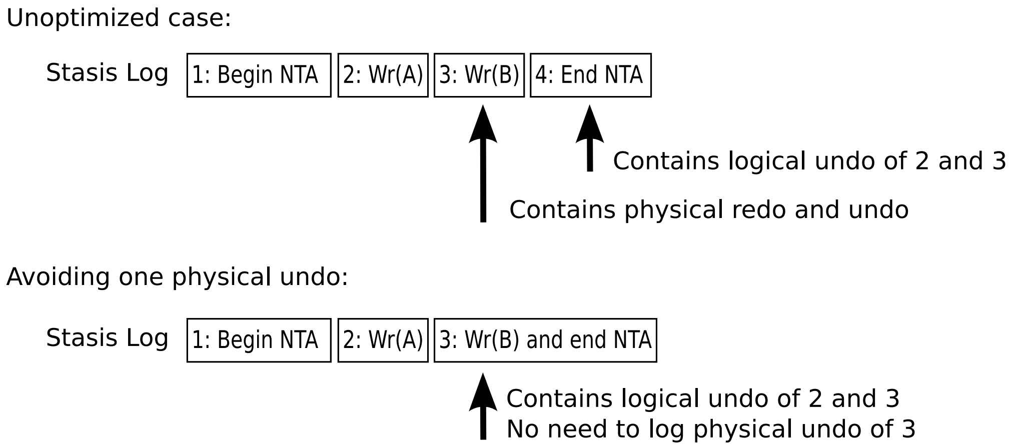

Unoptimized case:
Stasis Log
1: Begin NTA
2: Wr(A)
3: Wr(B)
4: End NTA
Contains logical undo of 2 and 3
Contains physical redo and undo
Avoiding one physical undo:
Stasis Log
1: Begin NTA
2: Wr(A)
3: Wr(B) and end NTA
Contains logical undo of 2 and 3
No need to log physical undo of 3

Figure 14.1. Reducing the overhead of nested top actions by removing one physical undo entry.

Because such issues are dealt with at a higher level, each nested top action can be represented using three pieces of information: the start LSN, the end LSN, and the logical undo data. Nested top actions within the same transaction may be nested, i.e., one may be initiated and completed while another is in progress. Conceptually, this forms a per-transaction stack. During forward operation, Stasis leaves maintenance of this stack to higher=level code; each time a nested top action is initiated Stasis returns a handle (which contains the start LSN) that must be passed into the call that completes the nested top action. Higher-level code may simply store the handle in the current stack frame. This adds minimal complexity, and has sufficed for every Stasis data structure implementation so far.

14.2.1 A note on log overheads and and optimizations

The primary disadvantage of nested top actions is log bandwidth. Consider an operation that overwrites a value in a B-Tree. It must write three values to the log: the old value of the entry for physical undo, the new value for physical redo, and a second copy of the old value for logical undo.

It is straightforward to eliminate the copy of the value written for the last physical undo. If the last physical entry is written atomically with the logical undo that completes the nested top action, then the physical undo will never be invoked. Omitting this physical undo reduces the amount of data logged at essentially no cost (Figure 14.1).

Unfortunately, extending this idea to multiple entries incurs significant overhead. The most straightforward approach simply prevents the log from being written until the nested top action completes. This blocks attempts to commit other transactions, and it blocks write-back of recently written pages. Such a scheme essentially implements a no-Steal/no-Force transactional storage system, except that log write-back is blocked in addition to page write-back. In the limit, if each transaction consisted of a single, long nested top action that performed this optimization, then, in order to commit a single transaction, Stasis would wait for all active transactions to complete, then commit them as a group (Figure 14.2). In systems with low concurrency or short transactions, this may be a worthwhile tradeoff, but it generally leads to poor resource utilization and unpredictable request latencies.

Ultimately, suppressing multiple physical undo entries trades off between concurrency and log

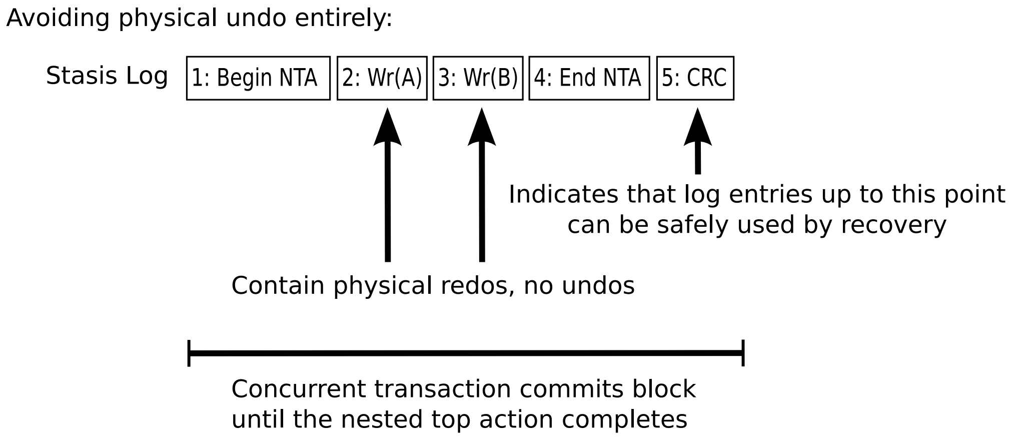

Avoiding physical undo entirely:
Stasis Log
1: Begin NTA
2: Wr(A)
3: Wr(B)
4: End NTA
5: CRC
Indicates that log entries up to this point can be safely used by recovery
Contain physical redos, no undos
Concurrent transaction commits block until the nested top action completes

Figure 14.2. Omitting multiple undo entries per nested top action blocks commits, leading to poor concurrency.

bandwidth, while suppressing a single entry per nested top action seems to be beneficial in all cases. Since the common case of most index operations can be reduced to a single physical operation, suppressing the final physical undo is likely the best general-purpose approach.

14.3 Write-back protocol

Page write-back serves a number of purposes. First, it allows dirty pages to be evicted eventually, making room for other data in the buffer pool. Second, it ensures that some prefix of the log has been applied to the page file, allowing the space taken up by that prefix of the log to be reclaimed.

If write-back fails to perform these tasks in a timely fashion, then Stasis will block application requests until sufficient numbers of appropriate pages have been written back. It is the responsibility of the *write-back policy* to schedule writes in a way that maximizes write throughput while avoiding such scenarios.

Although such policies are inherently complex, and depend on the application and runtime environment, they are essentially optimizations. This section focuses on correctness of write-back, which Stasis decouples from the actual write-back strategy.

In order to prevent bugs in write-back policies from violating write ahead, Stasis includes an abstraction called a *page handle* that is essentially a wrapper around a file handle. Page handles have access to the log and dirty page table. Before writing back a page, the page handle consults the dirty page table. If the page is clean, it ignores the request. (Races at higher levels can make it difficult to avoid issuing such requests without sacrificing concurrency.) Otherwise, it consults the log, and, if necessary, requests a force write for write-ahead. Finally, the page is written back.

Other than calls into other modules, this process is extremely lightweight. Furthermore, it is unlikely to change over time. This reflects one of Stasis' design goals. Understanding the complexity of concurrent transactional storage algorithms is difficult. Therefore, Stasis places the mechanisms that implement the recovery algorithm into dedicated, unchanging modules. This makes it easy to correctly evolve the rest of the system. This pattern shows up in other modules as well. One example is recovery; programming errors in the (otherwise straightforward) recovery code would have subtle correctness implications. At the same time, the recovery algorithm itself uses few resources; recovery times are dominated by other modules, such as redo and undo methods, the log, and the page file.

14.4 Log truncation protocol

Appending entries to the end of the log is straightforward; the only complications that can arise are due to running out of disk space during log write-back. Therefore, most recovery systems assume that the log is essentially unbounded, and over allocate space to ensure this is true.

Ignoring bounded disk space, it is always safe to append entries to the end of the log. However, truncating the log (deleting entries at the beginning of the log) to free up space is not so straightforward. First, Stasis consults the transaction table to determine when the first of the outstanding transactions began. That transaction could abort, which means we may need to access its log entries in the future. Next, Stasis consults the dirty page table to determine when the first dirty page was first modified. If both of these timestamps are high enough to warrant the expense, it then truncates the log up to the earlier of the two.

Depending on the size of the log (and amount of free disk space available), it may make sense for truncation to be more aggressive. For instance, it may preempt normal write-back in order to quickly write back old pages or it may apply backpressure on all but the oldest transactions. Unfortunately, while these techniques are essentially the last resort of a log-truncation thread running low on space, both degrade performance and introduce the possibility of deadlocks: if the system runs out of log space, it cannot make forward progress (or even abort transactions) until space in the log is freed.

Ultimately, there is no way for Stasis to guarantee that misbehaving applications will not force it into such situations. Therefore, avoiding these problems is left to external mechanisms such as admission control and to the system administrators responsible for provisioning storage.

Even with admission control and bounded-length transactions, it is possible that the system will run out of log space, simply because some page is always pinned by one thread or another, starving write-back, and eventually blocking log truncation. Stasis deals with this possibility by ensuring that page latches for write-back eventually have a higher priority than page latches for updates and reads. Therefore, once a page has been selected for write-back, the attempt will eventually succeed. Of course, this could lead to a deadlock, where write-back is blocked waiting to pin the page, and some transaction is blocked from unpinning the page until space is freed in the log.

If the system deadlocks because a pinned page is preventing log truncation, restarting Stasis will safely release the pin (although this is clearly an unattractive option). If a long-running transaction is the source of the problem, the only solution is to add more disk space to the system. Fortunately, admission control and appropriate write-back policies can prevent Stasis from running out of log space, effectively eliminating such deadlocks.

14.4.1 Media recovery

Although unlimited concurrency and transactions with unbounded length inevitably lead to logs of unbounded length, the problem can be completely avoided in traditional transaction processing environments. Such systems are designed to provide fault-tolerant storage within a single database instance. Today, many systems opt to simply replicate data rather than focus on building a reliable data store.

Nevertheless, established techniques for fault-tolerant, single-node storage exist. Note that these approaches do not provide highly available services; if the node fails, the data becomes temporarily unavailable. Instead, they focus on environments where such failures are rare and the data is valuable; no transactions should be lost in the event of a disk failure.

The basic approach is simple; a snapshot of the database's pages is periodically taken, and all log entries generated since the beginning of the snapshotting process are kept, rather than discarded during truncation. The page snapshot and the log are kept on a second, preferably physically distant, machine. Depending on the semantics desired, the backup's log may be updated once per transaction commit, or periodically according to some policy.

Crucially, the snapshot is taken one page at a time; recovery already handles the situations that may arise due to pages being written to the page file out of order; the same approaches apply equally well to media recovery.

Note that, in addition to tolerating drive failures, systems that make use of media recovery gracefully handle running out of space in primary (high-performance) storage. If we assume that the backup device is reliable (ie: two backups are kept), then the primary log may be truncated without regard to the state of the page file or the length of outstanding transactions.

It is also possible to migrate cold pages to the snapshot, assuming that the backup media performs reasonably well. Such hybrid approaches have been developed atop many types of storage systems; two examples not based upon transactional recovery are HP AutoRAID (146), and LHAM indexes (100). As hard drives continue to replace tape backup, such approaches become increasingly attractive.

Chapter 15

Conclusion and Future work

Because it supports a wide range of storage primitives and is designed to be extended, Stasis offers new opportunities and increased flexibility to application designers and storage researchers. Encouragingly, Stasis' users have diverse requirements. Some are interested in conventional high-concurrency, low-latency, update-in-place applications, while others would like to evaluate log-structured approaches. Such contradictory requirements would be difficult for existing monolithic systems to address. However, Stasis already provides appropriate mechanisms for both types of applications. So far, each new user has brought a new storage hardware stack to the table; a range of hardware-specific Stasis extensions are on the way.

We plan to continue to use Stasis as a platform for storage research. Flexible storage systems have been proposed as a solution to a number of open database engineering problems; such systems could serve as platforms for automated performance tuning research and facilitate benchmarks and evaluations of competing techniques. The current rapid evolution of storage hardware and application architectures has increased the need for such tools.

In their current state, segment-based recovery and Rose leave many questions unanswered. Both approaches appear to be well-equipped to leverage upcoming hardware advances, and both enable new approaches to distributed storage. However, they expose drastically different semantics and performance characteristics to applications, and are only two points in the design space that Stasis provides. As hardware continues to evolve, it will continue to change the tradeoffs between new and existing storage techniques.

This dissertation lays the groundwork for a new class of storage system designed to adapt to application requirements and changing hardware primitives. Stasis is an implementation of these ideas, and already has led to a number of successful research prototypes. As its implementation matures, we hope to use it to support practical, next-generation storage systems, and to use segments and Rose to build new classes of distributed storage systems.

Bibliography

[1] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006.

[2] D. J. Abadi, M. J. Cafarella, J. M. Hellerstein, D. Kossmann, S. Madden, and P. A. Bernstein, "How best to build web-scale data managers? A panel discussion," *PVLDB*, vol. 2, no. 2, p. 1647, 2009.

[3] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Transactions on Database Systems*, 1987.

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

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB*, 2001.

[6] J. E. Allchin and M. S. McKendry, "Synchronization and recovery of actions." in *PODC*, 1983, pp. 31–44.

[7] M. Armbrust, A. Fox, D. Patterson, N. Lanham, H. Oh, B. Trushkowsky, and J. Trutna, "SCADS: Scale-independent storage for social computing applications," in *Conference on Innovative Data Systems Research (CIDR)*, January 2009.

[8] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. King III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management." *ACM Transactions on Database Systems*, vol. 1, no. 2, pp. 97–137, 1976.

[9] J. Atwood, "Object-relational mapping is the Vietnam of computer science," http://www.codinghorror.com/blog/archives/000621.html.

[10] T. Barclay, J. Gray, and D. R. Slutz, "Microsoft terraserver: A spatial data warehouse," *SIGMOD*, vol. cs.DB/9907016, 2000.

[11] D. S. Batory, "Conceptual-to-internal mappings in commercial database systems," in *PODS*, 1984, pp. 70–78.

[12] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise, "GENESIS: An extensible database management system," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1711–1729, November 1988.

[13] D. S. Batory and C. C. Gotlieb, "A unifying model of physical databases," *ACM Transactions on Database Systems*, vol. 7, no. 4, pp. 509–539, 1982.

[14] A. Bechtolsheim, "Memory technologies for data intensive computing," in *HPTS*, 2009.

[15] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[16] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981.

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

[18] J. Bradley, "Operations data bases," in *VLDB*, 1978.

[19] ——, *File and Data Base Techniques.* CBS College Publishing, 1981.

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

[21] E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, 2001.

[22] ——, "Combining systems and databases: A search engine retrospective," in *Readings in Database Systems*, 4th ed., J. M. Hellerstein and M. Stonebraker, Eds., 2005.

[23] L. Cabrera, J. McPherson, P. Schwarz, and J. Wyllie, "Implementing atomicity in two systems: Techniques, tradeoffs, and experience," *TOSE*, vol. 19, no. 10, 1993.

[24] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson, and E. J. Shekita, "The architecture of the EXODUS extensible DBMS," in *OODS*, 1986, pp. 52–65.

[25] M. J. Carey, D. J. DeWitt, and J. F. Naughton, "The OO7 benchmark," in *SIGMOD*, 1993, pp. 12–21.

[26] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "Object and file management in the EXODUS extensible database system," in *VLDB*, 1986, pp. 91–100.

[27] D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Robie, and J. Simon, "XQuery update facility 1.0," 2009.

[28] S. Chandrasekaran and M. Franklin, "Streaming queries over streaming data," in *VLDB*, 2002.

[29] 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.

[30] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.

[31] S. Chaudhuri and G. Weikum, "Rethinking database system architecture: Towards a self-tuning RISC-style database system," in *VLDB*, 2000.

[32] E. Chavez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquin, “Searching in metric spaces,” *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321, September 2001.

[33] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, June 1970.

[34] R. A. Crus, “Data recovery in IBM Database 2,” *IBM Systems Journal*, vol. 23, no. 2, 1984.

[35] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *OSDI*, 2004.

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

[37] P. Druschel and L. L. Peterson, “Fbufs: A high-bandwidth cross-domain transfer facility,” in *SOSP*, 1993.

[38] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, Eds., *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[39] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[40] B. Fitzpatrick, “Distributed caching with memcached,” *Linux Journal*, August 2004.

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

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

[43] S. Ghemawat, H. Gobioff, and S. Leung, “The Google file system,” in *SOSP*, 2003, pp. 29–43.

[44] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent available partition-tolerant web services,” in *In ACM SIGACT News*, 2002, p. 2002.

[45] G. Graefe, “Sorting and indexing with partitioned B-Trees,” in *CIDR*, 2003.

[46] ——, “B-Tree indexes for high update rates,” *SIGMOD Rec.*, vol. 35, no. 1, pp. 39–44, 2006.

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

[48] J. Gray, “Long term storage trends and you,” 2006.

[49] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, “Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals,” *J. Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[50] J. Gray and G. Graefe, “The five-minute rule ten years later, and other computer storage rules of thumb,” *SIGMOD Record*, vol. 26, no. 4, pp. 63–68, 1997.

[51] J. Gray, P. Helland, P. O’Neil, and D. Shasha, “The dangers of replication and a solution,” in *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 173–182.

[52] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in *IFIP Working Conference on Modelling in Data Base Management Systems*, 1976, pp. 365–394.

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

[54] T. Greanier, "Serialization API," in *JavaWorld*, 2000.

[55] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. E. Culler, "Scalable, distributed data structures for Internet service construction." in *OSDI*, 2000, pp. 319–332.

[56] ——, "Scalable, distributed data structures for internet service construction," in *OSDI*, 2000, pp. 319–332.

[57] T. O. Group, *Distributed Transaction Processing: The XA Specification*, X/Open Company Ltd/, 1991.

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

[59] R. L. Haskin, Y. Malachi, W. Sawdon, and G. Chan, "Recovery management in QuickSilver," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 82–108, 1988.

[60] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems," in *VLDB*, 1995, pp. 562–573.

[61] "Hibernate," http://www.hibernate.org/.

[62] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt, "How to barter bits for chronons: Compression and bandwidth trade offs for database scans," in *SIGMOD*, 2007.

[63] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg, "McRT-Malloc: A scalable transactional memory allocator." in *ISMM*, 2006, pp. 74–83.

[64] *Intel X25-E SATA Solid State Drive.*

[65] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007.

[66] C. Jermaine, E. Omiecinski, and W. G. Yee, "The partitioned exponential file for database storage management," *The VLDB Journal*, vol. 16, no. 4, pp. 417–437, 2007.

[67] J. Kienzle, A. Strohmeier, and A. B. Romanovsky, "Open multithreaded transactions: Keeping threads and exceptions under control." in *WORDS*, 2001, pp. 197–205.

[68] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, 2009.

[69] W. Kim, J. F. Garza, N. Ballou, and D. Woelk, "Architecture of the ORION next-generation database system," *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[70] S. Kleiman, "Flash on compute servers," in *HPTS*, 2009.

[71] E. K. Kolodner and W. E. Weihl, "Atomic incremental garbage collection and recovery for a large stable heap." in *SIGMOD*, 1993, pp. 177–186.

[72] M. Kornacker, C. Mohan, J. Hellerstein, U. C. Berkeley, and U. C. Berkeley, "Concurrency and recovery in generalized search trees," in *SIGMOD*. ACM Press, 1997, pp. 62–72.

[73] G. E. Krasner and S. T. Pope, "A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system," *Journal of Object Oriented Programming*, 1988.

[74] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, June 1981.

[75] D. Kuo, "Model and verification of a data manager based on ARIES," *TODS*, vol. 21, no. 4, 1996.

[76] C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb, "The ObjectStore database system." *Communications of the ACM*, vol. 34, no. 10, pp. 50–63, 1991.

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

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

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

[80] B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp. 300–312, March 1988.

[81] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *VLDB*, 1980, pp. 224–232.

[82] G. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer, "Extensions to Starburst: Objects, types, functions, and rules," *Communications of the ACM*, vol. 34, no. 10, pp. 95–109, October 1991.

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

[84] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure." in *OSDI*, 2004, pp. 105–120.

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

[86] E. Markatos, "On caching search engine results," Institute of Computer Science, Foundation for, Tech. Rep., 1999.

[87] W. C. McGee, "Data base technology," *IBM J. Res. Develop*, vol. 25, pp. 505–519, 1981.

[88] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A database management system for semistructured data," *ACM SIGMOD Record*, vol. 26, no. 3, pp. 54–66, September 1997.

[89] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems*, 1984.

[90] E. Meijer, "Confessions of a used programming language salesman," *SIGPLAN Not.*, vol. 42, no. 10, pp. 677–694, 2007.

[91] "mlock(2)," in *Linux Programmers Manual*, 3.21 ed., 2008.

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

[93] ——, "A database perspective on Lotus Domino/Notes," in *SIGMOD Tutorial*, 1999.

[94] ——, "Implications of storage class memories (SCMs) on software architectures," in *HPTS*, 2009.

[95] 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*, vol. 17, no. 1, pp. 94–162, 1992.

[96] C. Mohan and F. Levine, *ARIES/IM: An efficient and high concurrency index management method using write-ahead logging.* ACM Press, 1992.

[97] J. E. B. Moss, *Nested transactions: An approach to reliable distributed computing.* MIT, 1985.

[98] ——, "Open nested transactions: Semantics and support," in *WMPI*, 2006.

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

[100] P. Muth, P. O'Neil, A. Pick, and G. Weikum, "Design, implementation, and performance of the LHAM log-structured history data access method," in *VLDB*, 1998.

[101] National Severe Storms Laboratory Historical Weather Data Archives, Norman, Oklahoma, from their Web site at http://data.nssl.noaa.gov.

[102] T. Neward, "The Vietnam of computer science," http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.

[103] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn, "Rethink the sync," in *OSDI*, 2006.

[104] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008, pp. 1099–1110.

[105] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[106] *OpenVMS Record Management Services Reference Manual*, June 2002.

[107] J. Ousterhout, "Ramcloud: Scalable high-performance storage entirely in dram," in *HPTS*, 2009.

[108] *Rails: Complete API.*

[109] R. Ramakrishnan and J. Gehrke, *Database Management Systems.* McGraw Hill, 2003.

[110] V. Raman and G. Swart, "How to wring a table dry: Entropy compression of relations and querying of compressed relations," in *VLDB*, 2006.

[111] H. T. Reiser, "ReiserFS," http://www.namesys.com.

[112] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *SOSP*, 1992.

[113] M. A. V. Salles, T. Cao, B. Sowell, A. J. Demers, J. Gehrke, C. Koch, and W. M. White, "An evaluation of checkpoint recovery for massively multiplayer online games." *PVLDB*, vol. 2, no. 1, pp. 1258–1269, 2009.

[114] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight recoverable virtual memory," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 33–57, February 1994.

[115] F. B. Schmuck and J. C. Wyllie, "Experience with transactions in QuickSilver," in *SOSP*, 1991, pp. 239–253.

[116] P. M. Schwarz and A. Z. Spector, "Synchronizing shared abstract types," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 223–250, April 1984.

[117] R. Sears and C. van Ingen, "Fragmentation in large object repositories," in *CIDR*, 2007.

[118] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, I. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979, pp. 23–34.

[119] M. Seltzer and M. Olsen, "LIBTP: Portable, modular transactions for UNIX," in *Usenix*, January 1992.

[120] E. J. Shekita and M. J. Zwilling, "Cricket: A mapped, persistent object store." in *POS*, 1990, pp. 89–102.

[121] R. Silaghi and A. Strohmeier, "Critical evaluation of the EJB transaction model," in *FIDJI*, 2002, pp. 15–28.

[122] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni, "Enabling transactional file access via lightweight kernel extensions," in *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09).* San Francisco, CA: USENIX Association, February 2009, pp. 29–42.

[123] R. P. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok, "Story book: An efficient extensible provenance framework," in *TAPP*, 2009.

[124] *SQL Server 2008 Documetation.* Microsoft, 2009, ch. Buffer Management.

[125] L. Stein, "How Perl saved the Human Genome Project," *Dr Dobb's Journal*, July 2001.

[126] M. Stonebraker, "Operating system support for database management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418, July 1981.

[127] ——, "The design of the postgres storage system," in *VLDB*, 1987.

[128] M. Stonebraker and G. Kemnitz, "The POSTGRES next-generation database management system," *Communications of the ACM*, vol. 34, no. 10, pp. 79–92, October 1991.

[129] M. Stonebraker, "Inclusion of new types in relational data base systems," in *ICDE*, 1986, pp. 262–269.

[130] M. Stonebraker and U. Çetintemel, "One size fits all: An idea whose time has come and gone," in *ICDE*, 2005, pp. 2–11.

[131] M. Stonebraker and J. M. Hellerstein, Eds., *Readings in Database Systems*, 4th ed. The MIT Press, 2005.

[132] M. Stonebraker and J. M. Hellerstein, "What goes around comes around," in *Readings in Database Systems*, 4th ed. The MIT Press, 2005.

[133] StorageReview.com, "Seagate barracuda 750es," http://www.storagereview.com/ST3750640NS.sr, 12 2006.

[134] X. Sun, R. Wang, B. Salzberg, and C. Zou, "Online B-tree merging," in *SIGMOD*, 2005.

[135] H. tai Chou, D. J. Dewitt, J. Geng, D. Wang, J. Zhang, and J. Xie, "An evaluation of buffer management strategies for relational database systems," in *VLDB*, 1985, pp. 127–141.

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

[137] *Distributed Transaction Processing: Reference Model*, The Open Group, 1996.

[138] S. Tsur and C. Zaniolo, "An implementation of GEM: supporting a semantic data model on a relational back-end." *SIGMOD Rec.*, vol. 14, no. 2, pp. 286–295, 1984.

[139] J. van den Bercken, J.-P. Dittrich, and B. Seeger, "javax.xxl: A prototype for a library of query processing algorithms," in *SIGMOD*, 2000, p. 588.

[140] W. Weihl and B. Liskov, "Implementation of resilient, atomic data types," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, pp. 244–269, April 1985.

[141] G. Weikum, C. Hasse, P. Broessler, and P. Muth, "Multi-level recovery," in *PODS*, 1990.

[142] G. Weikum and H.-J. Schek, "Architectural issues of transaction management in multi-layered systems," in *VLDB*, 1984, pp. 454–465.

[143] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *SIGMOD Rec.*, vol. 29, no. 3, pp. 55–67, 2000.

[144] S. J. White and D. J. DeWitt, "Quickstore: A high performance mapped object store," *VLDB J.*, vol. 4, no. 4, pp. 629–673, 1995.

[145] M. Widenius and D. Axmark, *MySQL Manual*.

[146] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," in *SOSP*, 1995.

[147] V.-F. Yong, J. F. Naughton, and J.-B. Yu, "Storage reclamation and reorganization in client-server persistent object stores," in *ICDE*, 1994, pp. 120–131.

[148] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI*, 2008, pp. 1–14.

[149] B. R. Zeeberg, J. Riss, D. W. Kane, K. J. Bussey, E. Uchio, W. M. Linehann, J. C. Barrett, and J. N. Weinstein, "Mistaken identifiers: Gene name errors can be introduced inadvertently when using Excel in bioinformatics," *BMC Bioinformatics*, 2004.

[150] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," in *SIGMOD*, 1997, pp. 159–170.

[151] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar RAM-CPU cache compression," in *ICDE*, 2006.