UNIVERSITY OF CALIFORNIA

Los Angeles

Data Storage Considered Modular

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Michael D. Mammarella

2010

The dissertation of Michael D. Mammarella is approved.

Lixia Zhang

Richard R. Muntz

Bruce L. Rothschild

Edward W. Kohler, Committee Chair

University of California, Los Angeles

2010

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Many people have helped me in the long journey towards the completion of this dissertation; surely any list I might come up with would accidentally omit someone. Nevertheless I would like to specifically acknowledge just a few of them here, for their outstanding support. My advisor Eddie Kohler has been a wonderful influence on my graduate career, and without him it would have been a very different, probably much less special, experience. In many ways he was just the advisor I needed, with the right combination of research interests, personality, and involvement in my work; he was there when I wanted his advice, and conveniently absent when I was less inclined to do any work. I would also like to acknowledge the many members of my lab, with whom I not only worked on actual research but also enjoyed many hours tinkering with hardware, software, ideas, and lab pets not quite as related to the work we were supposed to be doing. Finally, I would like to thank my parents, for only having asked a few times when I was going to graduate, and my loving wife, Christina, for so patiently waiting for me to do so.

Chapter 2 is a revision of "Generalized File System Dependencies," published in the Proceedings of the 2007 ACM Symposium on Operating Systems Principles (SOSP), ©ACM, 2007. It is joint work done with Chris Frost, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang, and was supported by the National Science Foundation, Microsoft Research, and Intel.

Chapter 3 is a revision of "Modular Data Storage with Anvil," published in the Proceedings of the 2009 ACM Symposium on Operating Systems Principles (SOSP), ©ACM,

2009. It is joint work done with Shant Hovsepian and Eddie Kohler, and was supported by the National Science Foundation, Microsoft Research, the Alfred P. Sloan Foundation, and Intel.

VITA

1982	Born, Stoneham, Massachusetts
2000–2003	Chancellor's Scholarship University of Massachusetts, Amherst
2004	Bachelor of Science, Computer Science and Mathematics University of Massachusetts, Amherst
2004–2005	Departmental Fellowship Computer Science Department University of California, Los Angeles
2005–2006	Teaching Assistant Computer Science Department University of California, Los Angeles
2006	Master of Science, Computer Science University of California, Los Angeles
2005-2010	Graduate Research Assistant Eddie Kohler, Computer Science Department University of California, Los Angeles

PUBLICATIONS

de los Reyes, A., Frost, C., Kohler, E., Mammarella, M., and Zhang, L. 2005. The KudOS Architecture for File Systems. Work in progress session, Twentieth ACM Symposium on Operating Systems Principles. (Brighton, United Kingdom, October 23-26, 2005). SOSP 05. ACM, New York, NY.

Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. 2007. Generalized File System Dependencies. In *Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles* (Stevenson, Washington, October 14-17, 2007). SOSP 07. ACM, New York, NY, 307–320.

Mammarella, M., Hovsepian, S., and Kohler, E. 2009. Modular Data Storage with Anvil. In *Proceedings of the Twenty-Second ACM Symposium on Operating Systems Principles* (Big Sky, Montana, October 11-14, 2009). SOSP 09. ACM, New York, NY, 147–160.

ABSTRACT OF THE DISSERTATION

Data Storage Considered Modular

by

Michael D. Mammarella

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2010

Professor Edward W. Kohler, Chair

Modularity is used in many software systems to increase code readability and reusability; it also brings additional configurability and extensibility. This dissertation focuses on *layered* modularity: systems divided into fine-grained, interacting modules. A layered system builds desired behavior from the combination of simple feature modules, making it easy for users to change behavior by layering modules or writing new ones. Despite the benefits of this sort of modularity, data storage systems, and in particular file systems and databases, are generally not designed in this way. For instance, existing systems cannot simply add metadata journaling to an arbitrary file system by connecting a journaling module to it; neither can they add on-disk data compression to an arbitrary database table format by connecting a compression module. One primary reason for this inflexibility may lie in the difficulty of achieving *consistency* in layered modules without harming performance. Consistency, in this context, is the property that the data on stable storage is at all times in a "good" state, for some definition of "good."

This work investigates two major types of data storage software systems, file systems and databases, and explores ways in which each can be made more modular while preserving the essential property of consistency. Two different approaches are taken: first, in a new file system implementation architecture, a new first-class object is introduced that allows modules to communicate write ordering requirements between each other while remaining only loosely coupled. Second, in a new database back end, all writing in the system is isolated to a small handful of dedicated modules, allowing most modules to deal exclusively with read-only data and to themselves be divided into read-only and create-only parts. These approaches have both been successful, and can even offer performance benefits by giving users and system designers more flexibility and control over their data.

These prototype systems demonstrate that storage systems can be decomposed into layered modular components while sacrificing neither consistency nor performance. This modularity also makes them much easier to reconfigure and customize, providing performance improvements and useful new features with minimal incremental effort.

Chapter 1

Introduction

Modularity is used in many software systems to increase code readability and reusability; it also brings additional configurability and extensibility. Dividing a large software system into appropriate modules can make new functionality easier to write, bugs easier to find, and development easier to parallelize. A system divided into fine-grained modules, each providing an optional feature, can make it easy for users to build desired configurations by layering modules together, and to add additional features by writing new modules.

Despite the benefits of this sort of modularity, data storage systems, and in particular file systems and databases, are generally not designed in this way. Most operating systems, while allowing each file system format to be implemented as a module so that several may coexist, provide no mechanism for finer-grained modularity. Features that in principle can be made independent of the file system format, like journaling, extended attributes, or user-controlled write ordering, are generally integrated into each file system format, if present at all. Databases are similar in design: many database management systems allow for selectable back ends, like MySQL's "storage engines," but each table can use only one back end. Features that could be separated, like data compression or other storage optimizations, must be integrated into the storage engines, and not every storage engine provides every feature. As a result, when choosing a file system or database storage engine, it is often necessary to prioritize desired features and pick the option that most closely meets the application's needs even though it may not meet them all. With more modular data storage systems, it might instead be possible to build exactly the desired system by layering together feature-providing modules.

In both of these cases, I believe that a primary reason existing systems are not more modular lies in the difficulty of achieving *consistency* in a modular way. Consistency, in this context, is the property that the data on stable storage is at all times in a "good" state, for some definition of "good." For instance, for a file system, consistency means that the file system can be used right away after recovering from an unclean shutdown, requiring at most a fast recovery procedure. (That is, a full file system check is not required, or can be safely postponed.) In a monolithic design, all consistency requirements are taken into account by the designer and implemented throughout the code by carefully controlling the order in which data is written to stable storage. This can be both a privilege and a burden, as the entire system both *can* and *must* be analyzed as a whole to implement that system correctly; for large, complex systems, this can easily lead to subtle bugs. For example, existing file systems do not correctly support storing a journaled file system in a file inside another file system: the carefully-controlled write ordering requirements of the nested file system are ignored by the outer file system as mere data writes. While each file system may be correct by itself, the "entire system" in this configuration is the combination of both, and the original analyses no longer apply. Despite the potential data loss consequences, nested file systems like this are increasingly popular, and are used to implement features like encrypted home directories in Mac OS X. Nevertheless, more modular designs that would correctly support this layering are difficult to achieve: the layered modules must cooperate to provide consistency, yet still be independent enough to reap the benefits of modularity.

My thesis is that data storage systems, and in particular file systems and database back ends, can be decomposed into modular components without substantial performance penalties; further, that doing so can dramatically increase the potential of these systems to be reconfigured and customized, providing performance improvements or useful new features with minimal incremental effort. In particular, I will show that efficient alternatives to synchronous disk access exist for such modular designs, while still providing the necessary semantics and the full benefits of modularity.

In Chapter 2, I describe Featherstitch, a file system implementation framework allowing many file system features to be implemented as separate modules, and explain the enabling mechanisms underlying its design. In particular, a new first-class object allows modules to communicate write ordering requirements between each other in a file-system-agnostic manner. I argue that the modularity thus gained comes with low overhead, and provides useful new features (both for file system implementors and application developers) as well as performance improvements in some cases. Featherstitch runs in the Linux kernel and supports several file systems and multiple different techniques for providing consistency. It has performance on par with native Linux file system implementations providing similar mechanisms, while supporting additional features like correctly nested file systems and user-specified write orderings.

In Chapter 3, I describe Anvil, a back-end storage system for a database composed of fine-grained modules which process and store data, and present many such modules from which a wide variety of different storage strategies can be built. To avoid consistency issues, most modules deal with read-only data (which has very simple consistency requirements), and are themselves divided into parts that access read-only data stores and parts that create those stores. I argue that, again, this modularity comes with low overhead, and allows the system to be easily reconfigured (by arranging existing modules) and customized (by writing new modules) to provide performance improvements. For instance, replacing the original B-tree-based back end in SQLite with Anvil can improve performance by up to a factor of 5.3 on a "conventional" workload, and Anvil can be configured to provide additional improvements for more specialized data and workloads.

The contributions of this dissertation include:

- the module system designs in Featherstitch and Anvil;
- several specific core modules in each that perform functions important for overall system functionality or performance;
- new abstractions in these systems that allow the modules to work together while remaining compartmentalized;
- the new functionality thus enabled, like exporting dependency specification to userspace in Featherstitch;
- and the discovery that systems like these can be built without substantial performance penalty, as evidenced by the prototypes themselves.

These two systems demonstrate that, indeed, modularity need not be eschewed in the design of file systems and database back ends for performance reasons. In fact, in both systems it provides new opportunities for performance improvements, while simultaneously making the designs simpler to understand, customize, and configure.

Chapter 2

Featherstitch

In this chapter, I describe Featherstitch, a modular file system framework allowing many file system features to be implemented as separate components. The primary mechanism enabling this modularity in Featherstitch is the *patch*, a first-class object representing a change to a cached disk block along with a set of *dependencies* on other patches. Through patches, modules can make their own changes to disk blocks, examine the changes made by other modules, or even modify the dependencies of those changes. Patches separate the specification of required disk write orderings from their enforcement, and allow loosely-coupled modules to accomplish tasks that would otherwise require close cooperation.

There are significant benefits to be gained by decomposing file system implementations into modules. For instance, features like journaling can be implemented generically, and added to any file system by connecting the modules together. Features like simulated extended attributes, or other metadata not supported by the underlying file system format, could also be written as separate modules. (Mac OS X, for example, simulates resource forks and other metadata like the source URL of downloaded files when it writes files to FAT file systems.) Both of these examples come with specific ordering requirements: journal logs and commit records must be written before file system updates, and hidden files storing simulated metadata must be updated at appropriate times relative to the files whose metadata they store. Existing systems must integrate these kinds of features into their monolithic file system implementations.

Another benefit of this design is that dependency enforcement can be localized into a dedicated module – the buffer cache. In Featherstitch, the buffer cache bases its decisions on which blocks to write, and when, on the structure of patch dependencies, ensuring that blocks are written to disk in an acceptable order. This frees other modules of this responsibility, allowing them to focus only on specifying the dependencies in the first place. With a "dumb" buffer cache whose writes to the disk can be controlled only by pinning pages into memory, the buffer cache must be micromanaged by other system components.

Patches and the separation of dependency specification and enforcement also make it possible to nest consistent file systems, and to extend dependency specification into userspace. Several operating systems support using a file stored in one file system as the "disk" on which another file system is stored; in Linux, the *loopback* driver allows such files to appear as block devices ("disks"). Existing storage systems treat the nested file system's block writes as merely application updates to the contents of a file in another file system – and, as a result, important ordering requirements may be lost and then violated. In Featherstitch, the nested file system's ordering requirements are represented using patches, which pass through the loopback module intact. Many applications also have consistency requirements similar to those that a file system or file system extension might require, but currently have only rudimentary APIs for making sure these requirements are fulfilled. In general, these APIs provide consistency only by providing *durability*; that is, by forcing immediate and synchronous disk writes of the requested data. In Featherstitch, these custom application consistency requirements can be directly communicated to the kernel's buffer cache, and enforced without application micromanagement.

The contributions of this work are the patch model and module system designs, the patchgroup mechanism that exports patches to applications, and several individual Featherstitch modules, such as the journal and buffer cache.

There were several notable challenges in developing Featherstitch. It took several iterations to refine the patch API to support rearranging existing patches, as the journal module does, while remaining efficient. The more invariants we could guarantee about patches, the more optimization opportunities became available – but enforcing too many invariants would restrict the API and make it not flexible enough for some modules. Likewise, we had to strike the right balance of power and safety in the userspace-visible patchgroup interface. Here, we had to be more restrictive, since we cannot count on the behavior of arbitrary user processes, but we still wanted to provide the maximum level of functionality while protecting the kernel. Another challenge was working with Linux's disk subsystem and on-disk caches to preserve the required block write orderings all the way to the physical disk media. Both Linux's disk scheduler (even, it turns out, the "no-op" scheduler) and on-disk write-back caches may reorder writes; changing that property can be anywhere from impossible to merely performance-degrading. Instead, we designed Featherstitch to work with these systems as they are, yet still without substantial performance penalties.

Featherstitch is joint work with Chris Frost, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. I designed most of its module system and introduced the patch concept. In practice, many optimizations are necessary for the system to be practical; these were mostly written by Chris Frost and will be part of his dissertation. Our paper on Featherstitch was published in SOSP 2007 [14], and work on the system is now complete.

2.1 Featherstitch Introduction

Write-before relationships, which require that some changes be committed to stable storage before others, underlie every mechanism for ensuring file system consistency and reliability from synchronous writes to journaling. (Journaling logs a set of intended changes before committing the changes, so that the changes become durable atomically even if the system crashes.) Featherstitch is a complete storage system[1] built on a concrete form of these relationships: a simple, uniform, and file system agnostic data type called the *patch*. Featherstitch's API design and performance optimizations make patches a promising implementation strategy as well as a useful abstraction. In particular, they allow the storage system to be decomposed into pluggable modules, each of which may produce, consume, examine, and modify patches to perform its function as part of the system.

A patch represents both a change to disk data and any *dependencies* that change has on other changes. Patches were initially inspired by BSD's soft updates dependencies [16], but whereas soft updates implement a particular type of consistency and involve many structures specific to the UFS file system [29], patches are fully general, specifying only how a range of bytes should be changed. This lets file system implementations specify a write-before relationship between changes without dictating a write order that honors that relationship. It lets storage system components examine and modify dependency structures independent of the file system's layout, possibly even changing one type of consistency into another. It also lets applications modify patch dependency structures, thus defining consistency policies for the underlying storage system to follow.

Since patches form a sort of "common language" among all the modules in the sys-

[1]In this chapter, we use the term "storage system" to refer to the software layer that implements file system services in an operating system. This helps to distinguish it from the term "file system," which we use to refer to individual (named) on-disk data layouts, and the specific modules within storage systems that implement them.

tem, most modules are compatible with most other modules and applications, allowing file system extensions and applications to work with many different file systems. File system implementers already find it difficult to provide consistency guarantees [29, 60] and implementations are often buggy [68, 69], a situation further complicated by file system extensions and special disk interfaces [10, 31, 39, 45, 47, 49, 66]. File system extension techniques such as stackable file systems [19, 41, 70, 71] leave consistency up to the underlying file system; any extension-specific ordering requirements are difficult to express at the VFS [23] layer. Although maintaining file system correctness in the presence of failures is increasingly a focus of research [12, 46], other proposed systems for improving file system integrity differ mainly in the kind of consistency they aim to impose, ranging from metadata consistency to full data journaling and full ACID transactions [15, 26, 65]. Some users, however, implement their own end-to-end reliability for some data and prefer to avoid *any* consistency slowdowns in the file system layer [63]. Patches can represent all these choices, and since they provide a common language for file systems and extensions to discuss consistency requirements, even combinations of consistency mechanisms can comfortably coexist.

Applications likewise have few mechanisms for controlling buffer cache behavior in today's systems, and robust applications, including databases, mail servers, and source code management tools, must choose between several mediocre options. They can accept the performance penalty of expensive system calls like `fsync` and `sync`, which request that the storage system fall back to slow synchronous writes, or use tedious and fragile sequences of operations that assume particular file system consistency semantics. *Patchgroups*, our example user-level patch interface, export to applications some of patches' benefits for kernel file system implementations and extensions. Modifying an IMAP mail server to use patchgroups required only localized changes, which were implemented by Chris Frost. The result both meets IMAP's consistency requirements on any reasonable patch-based file system and avoids the performance hit of full synchronization.

Production file systems use system-specific optimizations to achieve consistency without sacrificing performance; we had to improve performance in a general way. A naïve patch-based storage system scaled terribly, spending far more space and time on dependency manipulation than conventional systems. However, optimizations reduced patch memory and CPU overheads significantly. Room for improvement remains, particularly in system time, but Featherstitch outperforms equivalent Linux configurations on many of our benchmarks; it is at most 30% slower on others.

In this chapter, we describe patches abstractly, state their behavior and safety properties, and give examples of their use. We then describe the Featherstitch implementation, showing how it is decomposed into modules, and present several of the more interesting and useful modules. Finally, our evaluation compares Featherstitch and Linux-native file system implementations.

2.2 Related Work

Storage systems have often employed some form of modularity at least since the introduction of VFS [23], which made it possible to abstract the implementation of a file system from the interface to it. This allows multiple different file system drivers to coexist in the same kernel, but it leaves each file system driver as a monolithic entity. Some production systems, like Linux's JBD [60], make specific other features somewhat more modular: JBD is theoretically a file system agnostic journaling layer despite being used only by ext3. (It requires specific support to be added to each file system using it.) Other, and generally more experimental, systems have gone further, proposing stackable module software for file systems [19, 41, 48, 66, 67, 70, 71] that allows additional functionality to be layered on top of monolithic file system implementations. Featherstitch continues this line of work, and, by introducing patches, makes the implementations themselves divisible into smaller pieces. It also allows stacked features greater control of the way data is written to disk, making more kinds of features safe to implement this way.

Most modern file systems protect file system integrity in the face of possible power failure or crashes via journaling, which groups operations into transactions that commit atomically [44]. The content and the layout of the journal vary in each implementation, but in all cases, the system can use the journal to replay (or roll back) any transactions that did not complete due to the shutdown. A recovery procedure, if correct [68], avoids time-consuming file system checks on post-crash reboot in favor of simple journal operations.

Soft updates [16] are another important mechanism for ensuring post-crash consistency. Carefully managed write orderings avoid the need for synchronous writes to disk or duplicate writes to a journal; only relatively harmless inconsistencies, such as leaked blocks, are allowed to appear on the file system. As in journaling, soft updates can avoid scanning the file system after a crash to detect inconsistencies, although the file system must still be scanned in the background to recover leaked storage.

Patches naturally represent both journaling and soft updates, which we use as running examples throughout this chapter. In each case, our patch implementation extracts ad hoc orderings and optimizations into general dependency graphs, making the orderings potentially easier to understand and modify. Soft updates are in some ways a more challenging test of the patch abstraction: their dependencies are more variable and harder to predict, they are widely considered difficult to implement, and the existing FreeBSD implementation is quite optimized [29]. We therefore frequently discuss soft updates-like dependencies. This should not be construed as a wholesale endorsement of soft updates, which rely on a property (atomic block writes) that many disks do not provide, and which often require more seeks than journaling despite writing less data.

While journaling and soft updates are the most common file system consistency mechanisms currently in use, patches were designed to represent any write-before relationship. In Section 2.4.2, we present a module that uses patches to implement shadow paging-style techniques as found in write anywhere file layouts [20]; other arrangements, like ACID transactions [65], should also be possible.

CAPFS [62] and Echo [28] considered customizable application-level consistency protocols in the context of distributed, parallel file systems. CAPFS allows application writers to design plug-ins for a parallel file store that define what actions to take before and after each client-side system call. These plug-ins can enforce additional consistency policies. Echo maintains a partial order on the locally cached updates to the remote file system, and guarantees that the server will store the updates accordingly; applications can extend the partial order. Both systems are based on the principle that not providing the right consistency protocol can cause unpredictable failures, yet enforcing unnecessary consistency protocols can be extremely expensive. Featherstitch patchgroups generalize this sort of customizable consistency and bring it to disk-based file systems.

A similar application interface to patchgroups is explored in Section 4 of Burnett's dissertation [7]. However, the methods used to implement the interfaces are very different: Burnett's system tracks dependencies among system calls, associates dirty blocks with unique IDs returned by those calls, and duplicates dirty blocks when necessary to preserve ordering. Featherstitch tracks individual changes to blocks internally, allowing kernel modules a finer level of control, and only chooses to expose a userspace interface similar to Burnett's as a means to simplify the sanity checking required of arbitrary user-submitted requests. Additionally, our evaluation uses a real disk rather than trace-driven simulations.

Dependencies have been used in BlueFS [34] and xsyncfs [35] to reduce the aggregate performance impact of strong consistency guarantees. Xsyncfs's *external synchrony* provides users with the same consistency guarantees as synchronous writes. Application writes are not synchronous, however. They are committed in groups using a journaling design, but additional write-before relationships are enforced on *non-file system* communication: a journal transaction must commit before output from any process involved in that transaction becomes externally visible via, for example, the terminal or a network connection. Dependency relationships are tracked across IPC as well. Featherstitch patches could be used to link file system behavior and xsyncfs process dependencies, or to define cross-network dependencies as in BlueFS; this would remove, for instance, xsyncfs's reliance on ext3. Conversely, Featherstitch applications could benefit from the combination of strict ordering and nonblocking writes provided by xsyncfs.

Systems developed after Featherstitch have also realized the benefits of making the buffer cache aware of dependencies. For instance, Valor [50] provides transactional semantics at the file system level, but based on experience from previous systems, strives to modify the kernel as little as possible. Nevertheless, one of its two key kernel modifications is adding a simple form of dependencies to the kernel's buffer cache.

Some systems have generalized a *single* consistency mechanism. Linux's JBD, as mentioned above, is theoretically a reusable journaling layer suitable for use by any file system; however, the only file system that uses it is the one for which it was designed. XN enforces a variant of soft updates on any associated library file system, but still requires that those file systems implement soft updates again themselves [21].

Featherstitch adds to this body of work by designing a primitive that generalizes and makes explicit the write-before relationship present in many storage systems, and implementing a storage system in which that primitive is pervasive throughout. This allows Featherstitch' modules to be finer-grained, while layered, modular file system features can be implemented more safely; it also allows the extension of this primitive into userspace for use by applications.

2.3 Patches

Every change to stable storage in a Featherstitch system is represented by a *patch*. This section describes the basic patch abstraction and our implementation of that abstraction.

2.3.1 Disk Behavior

We first describe how disks behave in our model, and especially how disks commit patches to stable storage. Although our terminology originates in conventional disk-based file systems with uniformly-sized blocks, the model would apply with small changes to file systems with non-uniform blocks and to other media, including RAID and network storage.

We assume that stable storage commits data in units called **blocks**. All writes affect one or more blocks, and it is impossible to selectively write part of a block. In disk terms, a block is a sector or, for file system convenience, a few contiguous sectors.

A **patch** models any change to block data. Each patch applies to exactly one block, so a change that affects n blocks requires at least n patches to represent. Each patch is either **committed**, meaning written to disk; **uncommitted**, meaning not written to disk; or **in flight**, meaning in the process of being written to disk. The intermediate in-flight state models reordering and delay in lower storage layers; for example, modern disks often cache writes to add opportunities for disk scheduling. Patches are created as uncommitted. The operating system moves uncommitted patches to the in-flight state by writing their blocks to the disk controller. Some time later, the disk writes these blocks to stable storage and reports success. When the operating system receives this acknowledgment, it commits the relevant patches. Committed patches stay committed permanently, although their effects can be undone by subsequent patches. The sets $\boldsymbol{C}$, $\boldsymbol{U}$, and $\boldsymbol{F}$ represent all committed, uncommitted, and in-flight patches, respectively.

Patch p's block is written $blk[p]$. Given a block B, we write $\boldsymbol{C}_B$ for the set of committed patches on that block, or in notation $\boldsymbol{C}_B = \{p \in \boldsymbol{C} \mid blk[p] = B\}$. $\boldsymbol{F}_B$ and $\boldsymbol{U}_B$ are defined similarly.

Disk controllers in this model write in-flight patches one block at a time, choosing blocks in an arbitrary order. In notation:

1. Pick some block B with $\boldsymbol{F}_B \neq \varnothing$.
2. Write block B and acknowledge each patch in $\boldsymbol{F}_B$.
3. Repeat.

Disks perform better when allowed to reorder requests, so storage systems try to keep many blocks in flight. A block write will generally put all of that block's uncommitted patches in flight, but a storage system may, instead, write a *subset* of those patches, leaving some of them in the uncommitted state. As we will see, this is sometimes required to preserve write-before relationships.

We intentionally do not specify whether the underlying persistent storage device (e.g., the disk) writes blocks atomically. Some file system designs, such as soft updates, rely on block write atomicity, where if the disk fails while a block B is in flight, B contains either the old data or the new data on recovery. Many journal designs do not require this, and include recovery procedures that handle in-flight block corruption – for instance, if the memory holding the new value of the block loses coherence before the disk stops writing [58]. Since patches model the write-before relationships underlying these journal designs, patches do not provide block atomicity themselves, and a patch-based file system with soft updates-like dependencies should be used in conjunction with a storage device that provides block atomicity.

p	a patch
$blk[p]$	patch p's block
$\boldsymbol{C},\boldsymbol{U},\boldsymbol{F}$	the sets of all committed, uncommitted, and in-flight patches, respectively
$\boldsymbol{C}_B,\boldsymbol{U}_B,\boldsymbol{F}_B$	committed/uncommitted/in-flight patches on block B
$q \rightsquigarrow p$	q depends on p (p must be written before q)
$dep[p]$	p's dependencies: $\{x \mid p \rightsquigarrow x\}$
$q \rightarrow p$	q directly depends on p ($q \rightsquigarrow p$ means either $q \rightarrow p$ or $\exists x : q \rightsquigarrow x \rightarrow p$)
$ddep[p]$	p's direct dependencies: $\{x \mid p \rightarrow x\}$

Figure 2.1: Patch notation.

2.3.2 Dependencies

A patch-based storage system implementation represents write-before relationships using an explicit **dependency** relation. The disk controller and lower layers don't understand dependencies; instead, the storage system maintains dependencies and passes blocks to the controller in an order that preserves dependency semantics. Patch q *depends on* patch p, written $q \rightsquigarrow p$, when the storage system must commit q either after p or at the same time as p. (Patches can be committed simultaneously only if they are on the same block.) A file system should create dependencies that express its desired consistency semantics. For example, a file system with no durability guarantees might create patches with no dependencies at all; a file system wishing to strictly order writes might set $p_n \rightsquigarrow p_{n-1} \rightsquigarrow \cdots \rightsquigarrow p_1$. Circular dependencies among patches cannot be resolved and are therefore errors. For example, neither p nor q could be written first if $p \rightsquigarrow q \rightsquigarrow p$. (Although a circular dependency chain entirely within a single block would be acceptable, Featherstitch treats all circular chains as errors.) Patch p's *set* of dependencies, written $dep[p]$, consists of all patches on which p depends: $dep[p] = \{x \mid p \rightsquigarrow x\}$. Given a set of patches P, we write $dep[P]$ to mean the combined dependency set $\bigcup_{p \in P} dep[p]$.

The **disk safety property** formalizes dependency requirements by stating that the dependencies of all committed patches have also been committed:

$$dep[\boldsymbol{C}] \subseteq \boldsymbol{C}.$$

Thus, no matter when the system crashes, the disk is consistent in terms of dependencies. Since, as described above, the disk controller can write blocks in any order, a Featherstitch storage system must also ensure the independence of in-flight blocks. This is precisely stated by the **in-flight safety property:**

$$\text{For any block } B,\ \ dep[\boldsymbol{F}_B] \subseteq \boldsymbol{C} \cup \boldsymbol{F}_B.$$

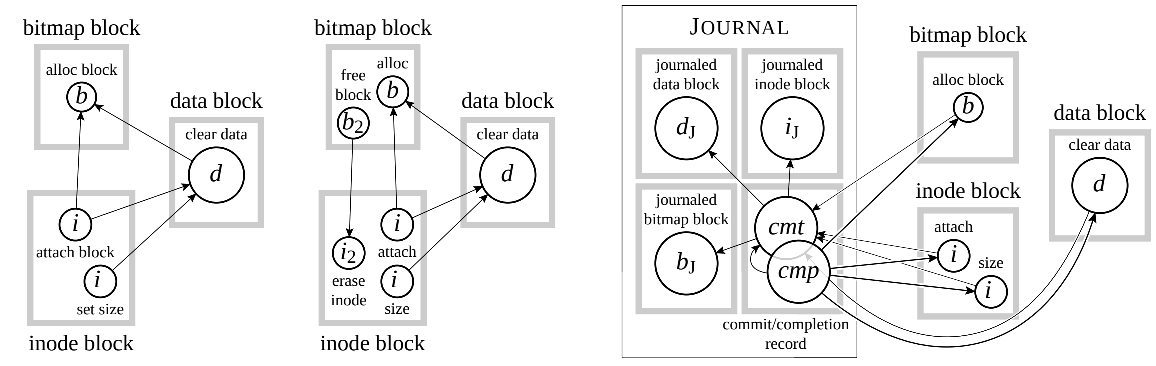
This implies that $dep[\boldsymbol{F}_B] \cap dep[\boldsymbol{F}_{B'}] \subseteq \boldsymbol{C}$ for any $B' \neq B$, so the disk controller can write in-flight blocks in any order and still preserve disk safety. To uphold the in-flight safety property, the buffer cache must write blocks as follows:

1. Pick some block B with $\boldsymbol{U}_B \neq \varnothing$ and $\boldsymbol{F}_B = \varnothing$.
2. Pick some $P \subseteq \boldsymbol{U}_B$ with $dep[P] \subseteq P \cup \boldsymbol{C}$.
3. Move each $p \in P$ to $\boldsymbol{F}$ (in-flight).

The requirement that $\boldsymbol{F}_B = \varnothing$ ensures that at most one version of a block is in flight at any time. Also, the buffer cache must eventually write *all* dirty blocks, a liveness property.

2.3.3 Dependency Implementation

The write-before relationship is transitive, so if $r \rightsquigarrow q$ and $q \rightsquigarrow p$, there is no need to explicitly store an $r \rightsquigarrow p$ dependency. To reduce storage requirements, a Featherstitch implementation maintains a subset of the dependencies called the *direct dependencies*. Each patch p has a corresponding set of direct dependencies $ddep[p]$; we say q *directly*



a) Adding a block (soft updates) **b)** . . . plus removing a file **c)** Adding a block (journaling)

Figure 2.2: Example patch arrangements for an ext2-like file system. Circles represent patches, shaded boxes represent disk blocks, and arrows represent direct dependencies. **a)** A soft updates order for appending a zeroed-out block to a file. **b)** A different file on the same inode block is removed before the previous changes commit, inducing a circular block dependency. **c)** A journal order for appending a zeroed-out block to a file.

depends on p, and write $q \rightarrow p$, when $p \in ddep[q]$. The dependency relation $q \rightsquigarrow p$ means that either $q \rightarrow p$ or $q \rightsquigarrow x \rightarrow p$ for some patch x.

Featherstitch maintains each block in its dirty state, including the effects of all uncommitted patches. However, each patch carries **undo data**, the previous version of the block data altered by the patch. If a patch p is not written with its containing block, the buffer cache *reverts* the patch, which swaps the new data on the buffered block and the previous version in the undo data. Once the block is written, the system will re-apply the patch and, when allowed, write the block again, this time including the patch. Some undo mechanism is required to break potential block-level dependency cycles, as shown in the next section. However, many of our optimizations avoid storing unnecessary undo data, greatly reducing memory usage and CPU utilization.

Figure 2.1 summarizes our patch notation.

2.3.4 Examples

This section illustrates patch implementations of two widely-used file system consistency mechanisms, soft updates and journaling. Our basic example extends an existing file by a single block – perhaps an application calls `ftruncate` to append 512 zero bytes to an empty file. The file system is based on Linux's ext2, an FFS-like[2] file system with inodes and a free block bitmap. In such a file system, extending a file by one block requires (1) allocating a block by marking the corresponding bit as "allocated" in the free block bitmap, (2) attaching the block to the file's inode, (3) setting the inode's size, and (4) clearing the allocated data block. These operations affect three blocks – a free block bitmap block, an inode block, and a data block – and correspond to four patches: b (allocate), i (attach), i' (size), and d (clear).

With soft updates, the final patch arrangement will consist of just these four patches, with some dependencies between them designed to preserve important invariants on the disk at all times. The correct dependencies are easy to determine, however, by following some simple rules also used in the original soft updates implementation. After going through the simple example, we also examine a variation that produces a block-level cycle, and contrast soft updates patches with the BSD implementation.

With journaling, on the other hand, we will end up with a much more complex-looking patch arrangement: since the example is so small, the extra patches for journaling seem like a large burden. In a larger transaction, a lower proportion of extra patches would be required – although there is insufficient space to depict such a transaction here. The journaling arrangement also naturally results in a block-level cycle, which we identify to help further clarify this important patch concept.

[2]Fast File System [30], an influential file system upon which many modern file system designs are based.

Soft updates Early file systems aimed to avoid post-crash disk inconsistencies by writing some, or all, blocks synchronously. For example, the write system call might block until all metadata writes have completed – clearly bad for performance. Soft updates provide post-crash consistency without synchronous writes by tracking and obeying necessary dependencies among writes. A soft updates file system orders its writes to enforce three simple rules for metadata consistency [16]:

1. "Never write a pointer to a structure until it has been initialized (e.g., an inode must be initialized before a directory entry references it)."
2. "Never reuse a resource before nullifying all previous pointers to it."
3. "Never reset the last pointer to a live resource before a new pointer has been set."

By following these rules, a file system limits possible disk inconsistencies to leaked resources, such as blocks or inodes marked as in use but unreferenced. The file system can be used immediately on reboot; a background scan can locate and recover the leaked resources while the system is in use.

These rules map directly to patches. Figure 2.2a shows a set of soft updates-like patches and dependencies for our block-append operation. Soft updates Rule 1 requires that $i \rightarrow b$. Rule 2 requires that d depend on the nullification of previous pointers to the block. A simple, though more restrictive, way to accomplish this is to let $d \rightarrow b$, where b depends on any such nullifications (there are none here). The dependencies $i \rightarrow d$ and $i' \rightarrow d$ provide an additional guarantee above and beyond metadata consistency, namely that no file ever contains accessible uninitialized data.

Figure 2.2b shows how an additional file system operation can induce a circular dependency among blocks. Before the changes in Figure 2.2a commit, the user deletes a one-block file whose data block and inode happen to lie on the bitmap and inode blocks used by the previous operation. Rule 2 requires the dependency $b_2 \rightarrow i_2$, but given this dependency and the previous $i \rightarrow b$, neither the bitmap block nor the inode block can be written first! Breaking the cycle requires rolling back one or more patches, which in turn requires undo data. For example, the system might roll back b_2 and write the resulting bitmap block, which contains only b. Once this write commits, all of i, i', and i_2 are safe to write; once *they* commit, the system can write the bitmap block again, this time including b_2.

Unlike Featherstitch, the BSD UFS soft updates implementation (which has been the default consistency mechanism in BSD for over a decade) represents each UFS operation by a different specialized structure encapsulating all of that operation's disk changes and dependencies. These structures, their relationships, and their uses are quite complex [29], and involve constant micromanagement by the file system code to ensure that appropriate block data is written to disk. After each write to the disk completes, callbacks process the structures and make pending, dependent changes to other cached disk blocks. In some cases, this mechanism even resulted in userspace-visible anomalies like incorrect link counts for directories when subdirectories had been recently removed, and required the addition of new "effective" metadata fields in other kernel structures to hide them.

Journal transactions A journaling file system ensures post-crash consistency using a write-ahead log. All changes in a transaction are first copied into an on-disk journal. Once these copies commit, a *commit record* is written to the journal, signaling that the transaction is complete and all its changes are valid. Once the commit record is written, the original changes can be written to the file system in any order, since after a crash the system can replay the journal transaction to recover. Finally, once all the changes have been written to the file system, the commit record can be erased, allowing that portion of the journal to be reused.

This process also maps directly to patch dependencies, as shown in Figure 2.2c. Copies of the affected blocks are written into the journal area using patches d_J, i_J, and b_J, each on its own block. Patch *cmt* creates the commit record on a fourth block in the journal area; it depends on d_J, i_J, and b_J. The changes to the main file system all depend on *cmt*. Finally, patch *cmp*, which depends on the main file system changes, overwrites the commit record with a completion record. Again, a circular block dependency requires the system to roll back a patch, namely *cmp*, and write the commit/completion block twice.

2.3.5 Patch Implementation

Our Featherstitch file system implementation creates patch structures corresponding directly to this abstraction. Functions like `patch_create_byte` create patches; their arguments include the relevant block, any direct dependencies, and the new data. Most patches specify this data as a contiguous byte range, including an offset into the block and the patch length in bytes. The undo data for very small patches (4 bytes or less) is stored in the patch structure itself; for larger patches, undo data is stored in separately allocated memory. In bitmap blocks, changes to individual bits in a word can have independent dependencies, which we handle with a special bit-flip patch type.

The implementation automatically detects one type of dependency. If two patches q and p affect the same block and have overlapping data ranges, and q was created after p, then Featherstitch adds an *overlap dependency* $q \to p$ to ensure that q is written after p. File systems need not detect such dependencies themselves.

For each block B, Featherstitch maintains a list of all patches with $blk[p] = B$. However, committed patches are not tracked; when patch p commits, Featherstitch destroys p's data structure and removes all dependencies $q \to p$. Thus, a patch whose dependencies have all committed appears like a patch with no dependencies at all. Each patch p maintains doubly linked lists of its direct dependencies and "reverse dependencies" (that is, all q where $q \to p$).

The implementation also supports *empty* patches, which have no associated data or block. Empty patches automatically commit when all of their dependencies commit, and can be used to "stand in" for future patches that have not yet been created by explicitly holding them in memory. The journal module does this; see Section 2.4.3. Empty patches can also be used shrink memory usage by representing densely bipartite sets of dependencies with a linear number of edges, instead of a quadratic number. This is useful for patchgroups; see Section 2.5. However, extensive use of empty patches adds to system time by requiring additional patch traversals. Our implementation uses empty patches infrequently, and in the rest of this section, patches are nonempty unless explicitly stated.

2.3.6 Optimizations

A naïve Featherstitch implementation creates many more patches, and allocates much more undo data, than would be reasonable in practice. For example, when 256 MiB of blocks are allocated in the untar benchmark described in Section 2.7, unoptimized Featherstitch allocates an additional 533 MiB, mostly for patches and undo data. Several optimizations, not part of this dissertation, were necessary in order to bring the performance of the system within reason. Most of the optimizations merge patches together or omit undo data when it is safe to do so, based on generic dependency analysis. Additional optimizations simplify Featherstitch's other main overhead, the CPU time required for the buffer cache to find a suitable set of patches to write. These optimizations apply transparently to any Featherstitch file system, and have dramatic effects on real benchmarks. For instance, they reduce memory overhead in the untar benchmark from 533 MiB to just 40 MiB.

2.3.7 Discussion

Optimizations can only do so much with bad dependencies. Just as having too few dependencies can compromise system correctness, having too many dependencies, or the wrong dependencies, can non-trivially degrade system performance. For example, in both the following patch arrangements, *s* depends on all of *r*, *q*, and *p*, but the left-hand arrangement gives the system more freedom to reorder block writes:

$$\begin{tikzcd} s \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left] & r & q & p \end{tikzcd} \qquad \begin{tikzcd} s \arrow[r] & r \arrow[r] & q \arrow[r] & p \end{tikzcd}$$

If *r*, *q*, and *p* are adjacent on disk, the left-hand arrangement can be satisfied with two disk requests while the right-hand one will require four. Although neither arrangement is much harder to code, in several cases we discovered that one of our file system implementations was performing slowly because it created an arrangement like the one on the right.

Care must also be taken to avoid accidental overlap dependencies, which can occur when patches are made larger than necessary. These additional dependencies enforce a chronological ordering among the overlapping patches, which would not have been required with smaller, independent patches. Patches that change one independent field at a time generally give the best results. For instance, inode blocks contain multiple inodes, and changes to two inodes should generally be independent; a similar statement holds for directories. Featherstitch will merge these patches when appropriate, but if they cannot be merged, minimal patches tend to cause fewer patch reversions and give more flexibility in write ordering.

File system implementations can also generate better dependency arrangements when they can detect that previous changes are being undone before being written to disk. For example, soft updates require that clearing an inode depend on nullifications of all corresponding directory entries, which normally induces dependencies from the inode onto the directory entries. However, if the file was recently created and its directory entry has yet to be written to disk, then a patch to remove the directory entry might merge with the patch that created it (which itself depends on the patch initializing the inode). In that case, there is no need for a dependency in *either* direction between the inode and directory entry blocks: the directory entry will *never* exist on disk. Leaving out these dependencies can speed up the system by avoiding block-level cycles, and the rollbacks and double writes they cause. The Featherstitch ext2 module implements several optimizations like this, significantly reducing disk writes, patch allocations, and undo data required when files are created and deleted within a short time. Although the optimizations are file system specific, the file system implements them using general properties, namely, whether two patches successfully merge.

Finally, block allocation policies can have a dramatic effect on the number of I/O requests required to write changes to the disk. For instance, soft updates-like dependencies require that data blocks be initialized before an indirect block references them. Allocating an indirect block in the middle of a range of file data blocks forces the data blocks to be written as two smaller I/O requests, since the indirect block cannot be written at the same time. Allocating the indirect block somewhere else allows the data blocks to be written in one larger I/O request, at the cost of (depending on readahead policies) a potential slowdown in read performance.

2.3.8 Debugging

We often found it useful to examine generated patch dependency graphs, both to find overly restrictive dependencies causing poor performance, and to identify missing dependencies that could violate consistency guarantees. To allow this, Featherstitch optionally logs patch operations to disk; a separate debugger inspects and generates graphs from these logs using the `graphviz` tool [13]. This facility enhanced the already-explicit dependency specification in Featherstitch by displaying the generated dependencies visually, making it easier to trace indirect dependencies and see when they did not exist. (It also made it easy to generate the graphs used in our paper.) Although the graphs could sometimes be daunting, involving hundreds or thousands of patches, they still proved invaluable in the development of the system and should help to simplify the implementation of new modules.

2.4 Modules

Now that we have described patches, we return to the original reason for introducing them: to allow a storage system to be divided into a collection of loosely cooperating modules. Since these modules explicitly specify their write ordering requirements with patches, multiple modules can cooperate to specify overall dependency requirements by passing patches back and forth. This allows implementers to write file system extensions, both providing and taking advantage of strong consistency guarantees, that would otherwise be difficult or impossible to implement. A typical Featherstitch configuration is composed of many such modules, which fall into three major categories: *block devices*, *common file systems*, and *low-level file systems*. Of these, the first two are much like existing storage system interfaces; the third, on the other hand, is unique to Featherstitch and helps divide file system implementations into smaller modules. Featherstitch provides all three types so that different modules can implement, and possibly also use, the interfaces that make sense for the features they provide.

Block device (BD) modules are closest to the disk, and have a fairly conventional block device interface with interfaces such as “read block” and “flush.” For example, the module that enforces patch dependencies, the buffer cache module, is of this type. The journal module is also a block device module; it adds journaling to whatever file system is run on it by transforming the incoming dependencies.

Common file system (CFS) modules live closest to the system call interface, and have an interface similar to VFS [23]. These modules generally do not deal with patches, but can be used to implement simple "stackable" file system extensions that do not require any specific dependencies (similar to [70, 71]). For instance, Featherstitch includes a case-insensitivity module of this type.

In between these two interfaces are modules implementing the low-level file system (L2FS) interface, which helps divide file system implementations into code common across block-structured file systems and code specific to a given file system layout. The L2FS interface has functions to allocate blocks, add blocks to files, allocate file names, and other file system micro-operations. Like BD functions, L2FS functions deal with patches, allowing file system extensions that need specific write orders for consistency to be implemented as L2FS modules. A generic CFS-to-L2FS module called UHFS ("universal high-level file system") decomposes familiar VFS operations like write, read, and append into L2FS micro-operations. Our ext2, UFS, and "waffle" file system modules implement the L2FS interface, and sit "on top of" block device modules. (This is where any other file systems would be implemented as well.)

Every L2FS and BD function that might modify the file system takes a `patch_t **p` argument. Before the function is called, `*p` is set to the patch, if any, on which the modification should depend; when the function returns, `*p` is set to some patch corresponding to the modification itself.

2.4.1 UHFS

The L2FS interface is very low-level, and abstracts, to the extent possible, only the *structure* of a given file system format – its layout on disk, but not how it is used. The ways in which those structures are used is often very similar across block-structured file systems, and in particular those influenced by the Fast File System [30]. The UHFS module implements high-level, VFS-like operations in terms of the lower-level L2FS interface, and is reusable for different file systems; for instance, we use the same UHFS module with both ext2 and UFS.

As an example, when appending data to a file, UHFS allocates a block using one L2FS function (`allocate_block`), writes the data to the new block, and appends the now-written block to the file's block list using a second L2FS function (`append_file_block`). It also knows how to hook the dependencies up for soft updates, so the underlying L2FS module need not understand how to hook up dependencies between those operations nor even when they might be used.

The UHFS module also deals with updating timestamps and other metadata, allowing L2FS modules between the file system format and UHFS to implement otherwise unsupported metadata (like Unix users and permissions on a FAT file system, for instance) by storing that metadata in hidden files.

2.4.2 ext2, UFS, and waffle

Featherstitch currently has L2FS modules that implement three file system types: Linux ext2, 4.2 BSD UFS (Unix File System, the modern incarnation of the Fast File System), and "waffle," which is a simple file system patterned after NetApp's WAFL [20]. The ext2 and UFS modules initially generate dependencies arranged according to the soft updates rules; other dependency arrangements, like journaling, are achieved by transforming these. To the best of our knowledge, our implementation of ext2 is the first to provide soft updates consistency guarantees. Unlike FreeBSD's soft updates implementation, once these modules set up dependencies, they no longer need to concern themselves with file system consistency: the block device subsystem will track and enforce dependencies.

These modules were not difficult to write, although they are fairly large. For the most part, they are merely reimplementations of the corresponding original versions in Linux and BSD, although somewhat simplified due to the UHFS module handling part of the work. The key property of these modules that distinguishes them from the original versions is that they generate patches, allowing other modules to examine, add to, and change the dependencies.

The waffle module, unlike the other two L2FS modules, generates dependencies arranged for shadow paging, where no block that is currently reachable from the file system root on disk may be written. Rather, a copy is made and updated, and all pointers to the block are updated (possibly recursively causing more blocks to be cloned). Periodically, the single root block is updated (in place) to point to the new tree of blocks, atomically switching from the old version of the file system to the new version. In this design, patch dependencies always point downwards, from the root to the leaves, so that the root can only be written once all the data to which it refers has also been written. In fact, the dependencies need not form a deep tree – the patch that updates the root block can just directly depend on all the others, giving the cache maximum flexibility in choosing which blocks to write first. As further evidence that L2FS file system modules are not difficult to write, we note that this module was written *during* the conference at which Featherstitch was first presented, in order to include a slide about it in the presentation.

2.4.3 Journal

The journal module is a block device module that automatically makes any block device journaled. It does this by transforming the incoming patches, presumably generated by a file system module like ext2, into patches implementing journal transactions. It uses a separate journal block device to store the journal, allowing many different possible configurations. For instance, the journal can be stored on a different partition on the same disk, or on a separate disk, or a network block device. The journal block device can even be a loopback block device (see §2.4.5) to a file within another file system – or the journaled file system itself to produce an "internal" journal. No special provisions are necessary to allow these configurations: patches convey all the required dependency information automatically.[3]

Modified blocks are copied into the journal device by creating new patches. A commit record patch is also created that depends on these other journal device patches; the original patches are in turn altered to depend on the commit record. Any soft updates-like dependencies among the original patches are removed, since they are not needed when the journal handles consistency; however, the journal does obey user-specified dependencies, in the form of patchgroups (see §2.5). Finally, a completion record, which overwrites the commit record, is created depending on the original patches. This arrangement is also described in Section 2.3.4 and depicted in Figure 2.2.

The journal format is similar to ext3's [60]: a transaction contains a list of block numbers, the data to be written to those blocks, and finally a single commit record. Although the journal modifies existing patches' direct dependencies, it ensures that any new dependencies do not introduce block-level cycles. (This is a statically worked out guarantee based on careful analysis of the code involved. It is not checked at runtime, unless specific debugging options are enabled.)

As in ext3, transactions are required to commit in sequence. The journal module sets each commit record to depend on the previous commit record, and each completion record to depend on the previous completion record. This allows multiple outstanding transac-

[3]The journal module does however have to detect recursive calls into itself, and not attempt to journal the journal blocks, when an internal journal is in use.

tions in the journal, which benefits performance, but ensures that in the event of a crash, the journal's committed transactions will be contiguous.

Since the commit record is created at the end of the transaction, the journal module uses dependencies on a special empty patch explicitly held in memory to prevent file system changes from being written to the disk until the transaction is complete.

Our journal module prototype can run in full data journal mode, where every updated block is written to the journal, or in metadata-only mode, where only blocks containing file system metadata are written to the journal. It can tell which blocks are which by looking for a special flag on each patch set by the UHFS module.

We also provide several other modules that modify dependencies, including an "asynchronous mode" module that removes all dependencies, allowing the buffer cache to write blocks in any order. Using the journal or asynchronous mode modules, the same ext2 module can be used in asynchronous, soft updates, or journaled modes. This is particularly useful for testing, but also of practical utility, as in Linux (for instance) separate ext2 and ext3 modules are required to support just two of these modes – and the ext2 module is no longer widely used, increasing the probability of serious bugs going undiscovered.

2.4.4 Buffer Cache

The Featherstitch buffer cache both caches blocks in memory and ensures that modifications are written to stable storage in a safe order. Modules "below" the buffer cache – that is, between its output interface and the disk – are considered part of the "disk controller"; they can reorder block writes at will without violating dependencies, since those block writes will contain only in-flight patches. The buffer cache sees the complex consistency mechanisms that other modules define as nothing more than sets of dependencies among patches; it has no idea what consistency mechanisms it is implementing, if any. In some sense, it is the “core” of a working Featherstitch system: it makes unnecessary the ad-hoc, fragile, and obfuscated buffer cache micromanagement required with a “dumb” buffer cache, and replaces it with generic dependency enforcement performed by a dedicated module.

Our prototype buffer cache module uses a modified FIFO policy to write dirty blocks and an LRU policy to evict clean blocks. (Upon being written, a dirty block becomes clean and may then be evicted.) The FIFO policy used to write blocks is modified only to preserve the in-flight safety property: a block will not be written if none of its patches are ready to write. Once the cache finds a block with ready patches, it extracts all ready patches P from the block, reverts any remaining patches on that block, and sends the resulting data to the disk driver. The ready patches are marked in-flight and will be committed when the disk driver acknowledges the write. The block itself is also marked in-flight until the current version commits, ensuring that the cache will wait until then to write the block again.

As a performance heuristic, when the cache finds a writable block n, it then checks to see if block $n+1$ can be written as well. It continues writing increasing block numbers until some block is either unwritable or not in the cache. This simple optimization greatly improves I/O wait time, since the I/O requests are merged and reordered in Linux’s elevator scheduler. Nevertheless, there may still be important opportunities for further optimization: for example, since the cache will write a block even if only one of its patches is ready, it can choose to revert patches unnecessarily when a different order would have required fewer writes.

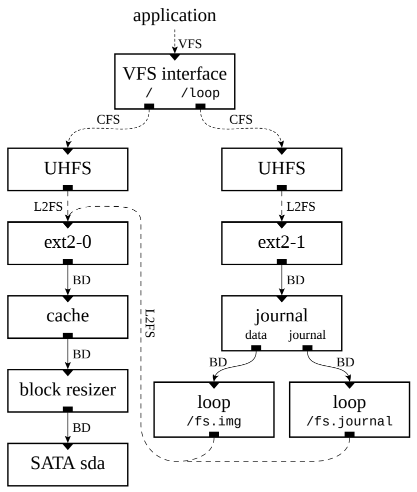


Figure 2.3: A running Featherstitch configuration. */* is a soft updates file system on a SATA drive; */loop* is an externally journaled file system on loop devices.

2.4.5 Loopback

The Featherstitch loopback module demonstrates how pervasive support for patches can implement previously unfamiliar dependency semantics. Like Linux's loopback device, it provides a block device interface that uses a file in some other file system to store its data. Unlike Linux's block device, consistency requirements on this block device are obeyed by the underlying file system. The loopback module forwards incoming dependencies to its underlying file system. As a result, the file system will honor those dependencies and preserve the nested file system's consistency policies, even if it would normally provide no consistency guarantees for file data (for instance, if it used metadata-only journaling).

Figure 2.3 shows an example configuration using two instances of the loopback module. A file system image is mounted with an external journal, both of which are loopback block devices stored on the root file system (which uses soft updates). The journaled file system's ordering requirements are sent through the loopback module as patches, maintaining dependencies across boundaries that might otherwise lose that information. Most systems cannot enforce consistency requirements through loopback devices this way –

unfortunate, as file system images are becoming popular tools in conventional operating systems, used for example to implement encrypted home directories in Mac OS X. In the event of a system failure, even though the outer file system might safely recover, the inner file system could require a full file system check before it would be safe to use.

The loopback module actually does very little. The only real work it does is to translate block device block numbers into file offsets within the backing file. In fact, to *not* forward the patch dependencies would be extra work. It is one of the simplest modules in Featherstitch, showing how Featherstitch's modular design and patches make it easy to write modules that enable entirely new classes of consistent file system configurations.

2.5 Patchgroups

Currently, robust applications can enforce necessary write-before relationships, and thus ensure the consistency of on-disk data even after system crash, in only limited ways: they can force synchronous writes using `sync`, `fsync`, or `sync_file_range`, or they can assume particular file system implementation semantics, such as journaling. With the patch abstraction, however, a process might specify just dependencies; the storage system could use those dependencies to implement an appropriate ordering. This approach assumes little about file system implementation semantics, but unlike synchronous writes, the storage system can still buffer, combine, and reorder disk operations.

This section describes *patchgroups*, an example API for extending patches to userspace. Applications *engage* patchgroups to associate them with subsequent file system changes; dependencies are defined among patchgroups. A parent process can set up a dependency structure that its child process will obey unknowingly. Patchgroups can apply to any file system, and even raw block device writes, as they are implemented as a module within Featherstitch. Just as patches allow Featherstitch to be broken into modules, patchgroups

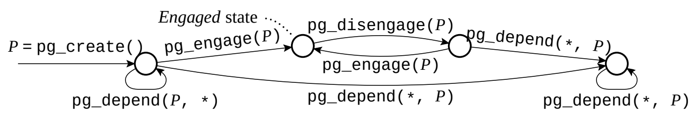


Figure 2.4: Patchgroup lifespan.

should enable applications with specific consistency requirements to be made more modular as well.

In this section we describe the patchgroup abstraction and apply it to three robust applications.

2.5.1 Interface and Implementation

Patchgroups encapsulate sets of file system operations into units among which dependencies can be applied. The patchgroup interface is as follows:

```
typedef int pg_t;
pg_t  pg_create(void);
int   pg_depend(pg_t Q, pg_t P); /* adds Q ⇝ P */
int   pg_engage(pg_t P);
int   pg_disengage(pg_t P);
int   pg_sync(pg_t P);
int   pg_close(pg_t P);
```

Each process has its own set of patchgroups, which are currently shared among all threads. The call `pg_depend(`*Q*`, `*P*`)` makes patchgroup *Q* depend on patchgroup *P*: all patches associated with *P* will commit prior to any of those associated with *Q*. *Engaging* a patchgroup with `pg_engage` causes subsequent file system operations to be associated with that patchgroup, until it is *disengaged*. Any number of patchgroups can be engaged at once; file system operations will be associated with all currently engaged patchgroups. `pg_sync` forces an immediate write of a patchgroup to disk. `pg_create` creates a new patchgroup and returns its ID, while `pg_close` disassociates a patchgroup ID from the underlying patches which implement it.

Whereas Featherstitch modules are presumed to not create cyclic dependencies, the kernel cannot safely trust user applications to be so well behaved, so the patchgroup API makes cycles unconstructable. Figure 2.4 shows when different patchgroup dependency operations are valid. As with patches themselves, all a patchgroup's direct dependencies are added first. After this, a patchgroup becomes engaged (allowing file system operations to be associated with it) zero or more times; however, once a patchgroup *P* gains a dependency via `pg_depend(*, P)`, it is sealed and can never be engaged again. This prevents applications from using patchgroups to hold dirty blocks in memory: *Q* can depend on *P* only once the system has seen the complete set of *P*'s changes.

Patchgroups and file descriptors are managed similarly – they are copied across `fork`, preserved across `exec`, and closed on `exit`. This allows existing, unaware programs to interact with patchgroups, in the same way that the shell can connect pipe-oblivious programs into a pipeline. For example, a `depend` program could apply patchgroups to unmodified applications by setting up the patchgroups before calling `exec`. The following command line would ensure that `in` is not removed until all changes in the preceding `sort` have committed to disk:

```
depend "sort < in > out" "rm in"
```

Without the patchgroup interface, an explicit `fsync` would be required after writing `out` (and before removing `in`) in order to achieve comparable consistency semantics. Further, it would force `out` to be written immediately, which in many cases may not be required and can hurt performance.

Patchgroups are implemented within Featherstitch by a special L2FS module. It uses several custom hooks into the rest of the kernel be notified when processes fork and exit, and registers an `ioctl` handler on a control device with which it implements the patchgroup user interface. These parts of the patchgroup implementation are global, and shared between all instances of the module, so that multiple instances can be active at once. Each

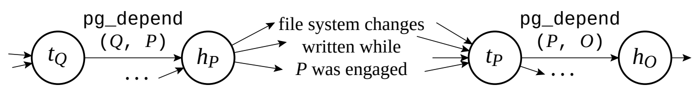


Figure 2.5: How patchgroups are implemented in terms of patches (simplified). Empty patches h_P and t_P bracket file system patches created while patchgroup P is engaged. `pg_depend` connects one patchgroup's t patch to another's h.

patchgroup corresponds to a pair of containing empty patches, and each inter-patchgroup dependency corresponds to a dependency between the empty patches. The patchgroup module inserts all file system changes made through it between the containing empty patches of any currently engaged patchgroups in the calling process. (That is, it creates incoming and outgoing dependencies between the file system changes and the containing empty patches.) Figure 2.5 shows an example patch arrangement for two patchgroups. (The actual implementation uses additional empty patches for bookkeeping.)

Patchgroups currently *augment* the underlying file system's consistency semantics, although a fuller implementation might let a patchgroup declare *reduced* consistency requirements as well.

2.5.2 Case Studies

We studied the patchgroup interface by adding patchgroup support to three relatively simple applications, and one much more complex application. The required changes to the simple applications, gzip, the Subversion [55] client, and the UW IMAP [61] server, were made by Chris Frost. Detailed discussion of the changes is left to his dissertation; however, we present performance results for the modified IMAP server in Section 2.7.4 as evidence that patchgroups can indeed help applications to achieve better performance. We chose these applications for their relatively simple and explicit consistency requirements, intending to test how well patchgroups can implement existing consistency mechanisms rather than create new mechanisms. The more complex application, Anvil, is described in detail in Chapter 3; our experience using patchgroups in its transaction library is described in Section 3.4.1. In all cases, patchgroups make the required guarantees explicit, and can be implemented on many types of file systems – the applications need not count on specific properties of particular file systems.

2.6 Implementation

The Featherstitch prototype implementation runs as a Linux 2.6 kernel module. It interfaces with the Linux kernel at the VFS layer and the generic block device layer. In between, a Featherstitch module graph replaces Linux's conventional file system layers. A small kernel patch informs Featherstitch of process fork and exit events as required to update per-process patchgroup state.

During initialization, the Featherstitch kernel module registers a VFS file system type with Linux. Each file system Featherstitch detects on a specified disk device can then be mounted from Linux using a command like `mount -t kfs kfs:`*`name`*` /mnt/point`. Since Featherstitch provides its own patch-aware buffer cache, it sets `O_SYNC` on all opened files as the simplest way to bypass the normal Linux cache and ensure that the Featherstitch buffer cache obeys all necessary dependency orderings.

Featherstitch modules interact with Linux's generic block device layer mainly via the kernel function `generic_make_request`. This function sends read or write requests to a Linux disk scheduler, which may reorder and/or merge the requests before eventually releasing them to the device. Writes are considered in flight as soon as they are enqueued on the disk scheduler. A callback notifies Featherstitch when the disk controller reports request completion; for writes, this commits the corresponding patches. The disk safety property requires that the disk controller wait to report completion until a write has reached stable storage. Most drives instead report completion when a write has reached the drive's volatile cache. Ensuring the stronger property could be quite expensive, requiring frequent barriers or setting the drive cache to write-through mode; either choice seems to prevent older drives from reordering requests. The solution is a combination of SCSI tagged command queuing (TCQ) or SATA native command queuing (NCQ) with either a write-through cache or "forced unit access" (FUA). TCQ and NCQ allow a drive to independently report completion for multiple outstanding requests, and FUA is a per-request flag that tells the disk to report completion only after the request reaches stable storage. Recent SATA drives handle NCQ plus write-through caching or FUA exactly as we would want: the drive appears to reorder write requests, improving performance dramatically relative to older drives, but reports completion only when data reaches the disk. We use a patched version of the Linux 2.6.20.1 kernel with good support for NCQ and FUA, and a recent SATA2 drive.

Our prototype has several performance problems caused by incomplete Linux integration. For example, writing a block requires copying that block's data whether or not any patches were undone, and our buffer cache currently stores all blocks in permanently-mapped kernel memory, limiting the buffer cache's maximum size.

2.7 Evaluation

To evaluate Featherstitch, we once again return to our basic goal in developing it: to divide storage system software into fine-grained modules while preserving consistency, and without a substantial performance penalty. Accordingly, we first evaluate the the performance of Featherstitch relative to Linux ext2 and ext3 using a variety of benchmarks, including the widely-used PostMark [22] and modified Andrew file system benchmarks. Next, we briefly evaluate the consistency properties and general correctness of the Featherstitch implementation by forcing spontaneous crashes and examining the state of the resulting disk images. Finally, we evaluate the performance of patchgroups using an IMAP server modified to use them and a simple benchmark moving many messages. This evaluation shows that a Featherstitch patch-based storage system has overall performance competitive with Linux, though using up to four times more CPU time; that Featherstitch file systems are consistent after system crashes; and that a patchgroup-enabled IMAP server outperforms the unmodified server on Featherstitch. It does not address the benefits of modularity in general or the extent to which Featherstitch's modular design in particular is beneficial; hopefully, this is at least to some extent evident from the configurations we use in the rest of this evaluation as well as those already presented. (Several individual modules, like the journal module and loopback module, also help to demonstrate it.)

2.7.1 Methodology

All tests were run on a Dell Precision 380 with a 3.2 GHz Pentium 4 CPU (with hyper-threading disabled), 2 GiB of RAM, and a Seagate ST3320620AS 320 GB 7200 RPM SATA2 disk. Tests use a 10 GiB file system and the Linux 2.6.20.1 kernel with the Ubuntu v6.06.1 distribution. Because Featherstitch only uses permanently-mapped memory, we disable high memory for all configurations, limiting the computer to 912 MiB of RAM. Only the PostMark benchmark performs slower due to this cache size limitation. All timing results are the mean over five runs.

To evaluate Featherstitch we ran four benchmarks. The *untar benchmark* untars and syncs the Linux 2.6.15 source code from the cached file `linux-2.6.15.tar` (218 MiB). The *delete benchmark*, after unmounting and remounting the file system following the untar benchmark, deletes the result of the untar benchmark and syncs. The *PostMark benchmark* emulates the small file workloads seen on email and netnews servers [22]. We use PostMark v1.5, configured to create 500 files ranging in size from 500 B to 4 MiB; perform

500 transactions consisting of file reads, writes, creates, and deletes; delete its files; and finally sync. The modified *Andrew benchmark* emulates a software development workload. The benchmark creates a directory hierarchy, copies a source tree, reads the extracted files, compiles the extracted files, and syncs. The source code we use for the modified Andrew benchmark is the Ion window manager, version 2-20040729.

2.7.2 Benchmarks and Linux Comparison

We benchmark Featherstitch and Linux for all four benchmarks, comparing the effects of different consistency models and comparing patch-based with non-patch-based implementations. Specifically, we examine Linux ext2 in asynchronous mode; ext3 in writeback and full journal modes; and Featherstitch ext2 in asynchronous, soft updates, metadata journal, and full journal modes. All file systems were created with default configurations, and all journaled file systems used a 64 MiB journal. Ext3 implements three different journaling modes, which provide different consistency guarantees. The strength of these guarantees is strictly ordered as “writeback $<$ ordered $<$ full.” Writeback journaling commits metadata atomically and commits data only after the corresponding metadata. Featherstitch metadata journaling is equivalent to ext3 writeback journaling. Ordered journaling commits data associated with a given transaction prior to the following transaction’s metadata, and is the most commonly used ext3 journaling mode. Doing this requires ensuring that blocks allocated during a transaction were not in use prior to the transaction – otherwise, if the transaction is interrupted before it commits, the previous uses of those blocks will be clobbered. While this concern is orthogonal to the use of patches, it does require that the block allocator be aware of transactions, or that the journal module can hook into the block allocator to ensure this; Featherstitch does not currently have either of these features and so does not provide ordered mode journaling. In all tests ext3 writeback and ordered jour-

System	**Untar**	**Delete**	**PostMark**	**Andrew**
Featherstitch ext2				
soft updates	**6.4 [1.3]**	**0.8 [0.1]**	**38.3 [10.3]**	**36.9 [4.1]**
meta journal	**5.8 [1.3]**	**1.4 [0.5]**	**48.3 [14.5]**	**36.7 [4.2]**
full journal	**11.5 [3.0]**	**1.4 [0.5]**	**82.8 [19.3]**	**36.8 [4.2]**
async	4.1 [1.2]	0.7 [0.2]	37.3 [6.1]	36.4 [4.0]
full journal	10.4 [3.7]	1.1 [0.5]	74.8 [23.1]	36.5 [4.2]
Linux				
ext3 writeback	**16.6 [1.0]**	**4.5 [0.3]**	**38.2 [3.7]**	**36.8 [4.1]**
ext3 full journal	**12.8 [1.1]**	**4.6 [0.3]**	**69.6 [4.5]**	**38.2 [4.0]**
ext2	4.4 [0.7]	4.6 [0.1]	35.7 [1.9]	36.9 [4.0]
ext3 full journal	10.6 [1.1]	4.4 [0.2]	61.5 [4.5]	37.2 [4.1]

Figure 2.6: Benchmark times (seconds). System CPU times are in square brackets. Safe configurations are **bold**, unsafe configurations are normal text.

naling modes performed similarly; since Featherstitch does not implement ordered mode, we report only writeback results. Full journaling commits data atomically.

There is a notable write durability difference between the default Featherstitch and Linux ext2/ext3 configurations: Featherstitch safely presumes a write is durable after it is on the disk platter, whereas Linux ext2 and ext3 by default presume a write is durable once it reaches the disk cache. However, Linux can write safely, by restricting the disk to providing only a write-through cache, and Featherstitch can write unsafely by disabling FUA. We distinguish safe (FUA or a write-through cache) from unsafe results when comparing the systems. Although safe results for Featherstitch and Linux utilize different mechanisms (FUA vs. a write-through cache), we note that Featherstitch performs identically in these benchmarks when using either mechanism.

The results are listed in Figure 2.6; safe configurations are listed in bold. In general, Featherstitch performs comparably with Linux ext2/ext3 when providing similar durability guarantees. Linux ext2/ext3 sometimes outperforms Featherstitch (for the PostMark test in journaling modes), but more often Featherstitch outperforms Linux. There are several possible reasons, including slight differences in block allocation policy, but the main point is that Featherstitch's general mechanism for tracking dependencies does not significantly degrade total time. Unfortunately, Featherstitch can use up to four times more CPU time than Linux ext2 or ext3. (Featherstitch and Linux have similar system time results for the Andrew benchmark, perhaps because Andrew creates relatively few patches even in the unoptimized case.) Higher CPU requirements are an important concern and, despite much progress in our optimization efforts, remain a weakness. Some of the contributors to Featherstitch CPU usage are inherent, such as patch creation, while others are artifacts of the current implementation, such as creating a second copy of a block to write it to disk; we have not separated these categories.

2.7.3 Correctness

In order to check that we had implemented the journaling and soft updates rules correctly, we implemented a Featherstitch module which crashes the operating system, without giving it a chance to synchronize its buffers, at a random time during each run of the above benchmarks. In Featherstitch asynchronous mode, after crashing, `fsck` nearly always reported that the file system contained many references to inodes that had been deleted, among other errors: the file system was corrupt. With our soft updates dependencies, the file system was always soft updates consistent: `fsck` reported, at most, that inode reference counts were higher than the correct values (an expected discrepancy after a soft updates crash). With journaling, `fsck` always reported that the file system was consistent after the journal replay.

2.7.4 Patchgroups

One of the benefits of Featherstitch's modularity is the ability to create modules that arrange patch dependencies to accurately reflect the desired write ordering, without imposing unnecessary overhead. The patchgroup module even extends this ability into userspace, where many applications can take advantage of this flexibility and avoid using mechanisms like `fsync`. To demonstrate the effectiveness of this ability, we evaluate the performance of an patchgroup-enabled UW IMAP mail server by benchmarking moving 1,000 messages from one folder to another. To move the messages, the client selects the source mailbox (containing 1,000 2 KiB messages), creates a new mailbox, copies each message to the new mailbox and marks each source message for deletion, expunges the marked messages, commits the mailboxes, and logs out.

Figure 2.7 shows the results for safe file system configurations, reporting total time, system CPU time, and the number of disk write requests (an indicator of the number of required seeks in safe configurations). We benchmark Featherstitch and Linux with the unmodified server (sync after each operation), Featherstitch with the patchgroup-enabled server (`pg_sync` on durable operations), and Linux and Featherstitch with the server modified to assume and take advantage of fully journaled file systems (changes are effectively committed in order, so sync only on durable operations). Only safe configurations are listed; unsafe configurations complete in about 1.5 seconds on either system. Featherstitch meta and full journal modes perform similarly; we report only the full journal mode. Linux ext3 writeback, ordered, and full journal modes also perform similarly; we again report only the full journal mode. Using an `fsync` per durable operation (CHECK and EXPUNGE) on a fully journaled file system performs similarly for Featherstitch and Linux; we report the results only for Linux full journal mode.

In all cases Featherstitch with patchgroups performs better than Featherstitch with `fsync` operations. Fully journaled Featherstitch with patchgroups performs at least as well as all other (safe and unsafe) Featherstitch and all Linux configurations, and is 11–13 times faster than safe Linux ext3 with the unmodified server. Soft updates dependencies are far slower than journaling for patchgroups: as the number of write requests indicates,

Implementation	**Time (sec)**	**# Writes**
Featherstitch ext2		
soft updates, `fsync` per operation	65.2 [0.3]	8,083
full journal, `fsync` per operation	52.3 [0.4]	7,114
soft updates, patchgroups	28.0 [1.2]	3,015
full journal, patchgroups	1.4 [0.4]	32
Linux ext3		
full journal, `fsync` per operation	19.9 [0.3]	2,531
full journal, `fsync` per durable operation	1.3 [0.3]	26

Figure 2.7: IMAP benchmark: move 1,000 messages. System CPU times shown in square brackets. Writes are in number of requests. All configurations are safe.

each patchgroup on a soft updates file system requires multiple write requests, such as to update the destination mailbox and the destination mailbox's modification time. In contrast, journaling is able to commit a large number of copies atomically using only a small constant number of requests. The unmodified `fsync`-per-operation server generates dramatically more requests on Featherstitch with full journaling than Linux, possibly indicating a difference in `fsync` behavior. The last line of the table shows that synchronizing to disk once per durable operation with a fully journaled file system performs similarly to using patchgroups on a journaled file system. However, patchgroups have the advantage that they work equally safely, and efficiently, for other forms of journaling.

With the addition of patchgroups UW IMAP is able to perform mailbox modifications significantly more efficiently, while preserving mailbox modification safety. On a metadata or fully journaled file system, UW IMAP with patchgroups is 97% faster at moving 1,000 messages than the unmodified server achieves using `fsync` to ensure its write ordering requirements.

2.7.5 Evaluation Summary

We find that Featherstitch has competitive performance on several benchmarks, despite the additional effort required to maintain patches; that CPU time remains an optimization opportunity; that applications can effectively define consistency requirements with patchgroups that apply to many file systems; and that the Featherstitch implementation correctly implements soft updates and journaling consistency. Our results indicate that even a patch-based prototype can implement different consistency models with reasonable cost.

2.8 Summary

Consistency is an overall property of a storage system, involving every possible write order that might be generated by any code in the system. Existing storage system implementations lack fine-grained modularity, and one major reason is that it is difficult to achieve consistency in a modular way. Featherstitch patches provide a new way for storage system implementations to formalize the “write-before” relationship among buffered changes to stable storage, making the building blocks of consistency into first-class objects. This separates the specification and enforcement of the desired dependencies, and allows many modules and even user applications to cooperate loosely while providing strong consistency guarantees. The resulting modular design also simplifies the implementation of consistency mechanisms like journaling and soft updates, as the latter is made explicit and easier to understand within each file system module, and the former can even be implemented as a separate module. This design also allows file systems to nest while providing correct consistency semantics, using a loopback module that forwards dependencies from the inner file system to the containing file system. The enforcement of dependencies by a dedicated module allows user applications to specify custom dependencies, via the patchgroup module, in addition to any generated within the storage system. This provides the buffer cache more freedom to reorder writes without violating the application’s needs, while simultaneously freeing the application from having to micromanage writes to disk. We present results for an IMAP server modified to take advantage of this feature, and show that it can significantly reduce both the total time and the number of writes required for our benchmark. Thanks to several optimizations, the performance of our prototype is usually at least as fast as Linux when configured to provide similar consistency guarantees, although in some cases it still requires improvement.

Nevertheless, the approach taken in Featherstitch can be a difficult one to apply to existing systems: while it may be easier to implement file systems within Featherstitch, that work has already been done for those systems. Even if maintaining Featherstitch versions might be easier, the initial cost of rewriting large amounts of well-tested code may be too high to justify. Further, Featherstitch adds a large dynamic memory allocation burden inside the kernel, especially when patchgroups – perhaps the feature most likely to help justify the reimplementation cost – are involved. Realizing these shortcomings of the Featherstitch approach to dividing a data storage system into modular components, but sensing that we were on the right track, we decided to explore another approach in the context of a different type of storage system. In the next chapter, rather than creating a new first-class data type to represent consistency requirements inside a file system implementation, we isolate the consistency-sensitive portions of database back end to only a very few modules, leaving the rest with no consistency responsibilities at all. As we will see, this approach is quite successful, and it is also easier to see how these techniques could be applied to existing systems.

Chapter 3

Anvil

Databases are used to store a tremendous amount of structured data, yet provide application designers little control over how that data is actually stored and processed. What configurable options exist generally control cache sizes and other simple system parameters, rather than the actual storage strategies involved. Query language features allow the specification of data types and indices, which offer high-level control over how queries will be processed, but do not offer lower-level control over data representation and layout. Some databases do allow additional "storage engine" modules to be loaded, but these, much like file system modules, are internally monolithic and only modular in that several can coexist.

I present Anvil, a back-end storage system for databases composed of fine-grained modules which process and store data. The modules, called *dTables*, implement a key-value interface, and also provide iterators allowing in-order traversal of the elements. Most of the data in a Anvil system is read-only; special dTables overlay writable dTables and read-only dTables to provide the illusion of fully-writable stores. This separation is central to Anvil's modular design, as it not only splits functionality among dTables but also the functionality within each dTable: dTables for read-only data (the most common type of dTable) can be further subdivided into code to read and code to create data stores. This makes writing new dTables easy, both because the separate code is easier to understand and also because it eliminates most transaction-related consistency concerns.

Further, just as the “common language” of patches allows Featherstitch modules to communicate with one another, the pervasive iterator mechanism used to convey data between dTables provides an important communication channel within Anvil. For instance, it makes possible “specialized” dTables, which store data more efficiently by limiting what data can be stored. (A specialized dTable that only stores 4-byte integers need not store data length information, for example.) Anvil allows these specialized dTables to be used in combination with other, more general, dTables to provide a fully general store that is optimized for common-case data. Finally, the separation of read-optimized and write-optimized dTables also allows Anvil to very effectively take advantage of the properties of disks, and even other storage technologies like flash memory, which are much faster at sequential access than random access.

Together, all of these characteristics of the dTable design allow the task of storing large amounts of data to be accomplished by a collection of loosely cooperating modules. While a major goal of Anvil is to make writing new dTables simple, it also includes a number of built-in dTables which can be arranged to store data in many traditional and recently-proposed new ways.

The contributions of this work are the fine-grained, modular dTable design, including an *iterator* facility whose *rejection* feature simplifies the construction of specialized tables; several core dTables that overlay and manage other dTables; and the Anvil implementation, which demonstrates that fine-grained database back end modularity need not carry a severe performance cost.

There were several notable challenges in developing Anvil. Correctly implementing transactions atop Linux ext3 proved to be quite tricky; ext3 does not officially provide any mechanisms to support this other than `fsync`, which forces synchronous disk access. Anvil uses knowledge of ext3's implementation to "trick" it into providing the necessary support by carefully ordering its system calls, but getting this to work correctly took several iterations and was very error-prone. Another challenge was choosing interfaces flexible enough to accommodate many different kinds of dTables while remaining simple and easy to implement. For instance, some specialized dTables cannot store all possible values, and may need to work with other dTables to store the values they cannot themselves store; this complication, however, should not make it more difficult to implement unrelated dTables. Other challenges included optimizing overlay iteration to avoid expensive key comparisons, and designing and implementing the small handful of dTables that handle consistency concerns. Also challenging, but in an entirely different way, was adapting SQLite to use Anvil as a back end; while incredibly well-documented, it nevertheless can be difficult to understand and modify.

Anvil is joint work with Shant Hovsepian and Eddie Kohler. I wrote most of Anvil; Shant has worked on a few parts of the system, but primarily was responsible for getting several of the benchmarks working.

3.1 Anvil Introduction

Database management systems offer control over how data is physically stored, but in many implementations, ranging from embeddable systems like SQLite [51] to enterprise software like Oracle [38], that control is limited. Users can tweak settings, select indices, or choose from a short menu of table storage formats, but further extensibility is limited to coarse-grained interfaces like MySQL's custom storage engines [33]. Even recent specialized engines [9, 54] – which have shown significant benefits from data format changes, such as arranging data in columns instead of the traditional rows [17, 53] or compressing sparse or repetitive data [1, 64] – seem to be implemented monolithically. A user whose application combines characteristics of online transaction processing and data warehousing may want a database that combines storage techniques from several engines, but database systems rarely support such fundamental low-level customization.

We present Anvil, a modular, extensible toolkit for building database back ends. Anvil comprises flexible storage modules that can be configured to provide many storage strategies and behaviors. We intend Anvil configurations to serve as single-machine back-end storage layers for databases and other structured data management systems.

The basic Anvil abstraction is the *dTable*, an abstract key-value store. Some dTables communicate directly with stable storage, while others layer above storage dTables, transforming their contents. dTables can represent row stores and column stores, but their fine-grained modularity offers database designers more possibilities. For example, a typical Anvil configuration splits a single "table" into several distinct dTables, including a log to absorb writes and read-optimized structures to satisfy uncached queries. This design introduces opportunities for clean extensibility – for example, we present a Bloom filter dTable that can slot above read-optimized stores and improve the performance of nonexistent key lookup. It also makes it much easier to construct data stores for unusual or specialized types of data; we present several such specialized stores. Conventional read/write functionality is implemented by dTables that *overlay* these bases and harness them into a seamless whole.

Results from our prototype implementation of Anvil are promising. Anvil can act as a back end for a conventional, row-based query processing layer – here, SQLite – and for hand-built data processing systems. Though Anvil does not yet support fully concurrent access, many parts of it are already thread-safe to allow some specific kinds of concurrent access. Additionally, our evaluation demonstrates that Anvil's modularity does not significantly degrade performance. Anvil generally performs about as well as or better than existing back-end storage systems based on B-trees on "conventional" workloads while providing similar consistency and durability guarantees, and can perform better still when customized for specific data and workloads.

In this chapter, we describe Anvil's general design and the Anvil transaction library, which provides the rest of the system with transactional primitives. We then describe many of Anvil's individual dTables, giving examples of their use to store different kinds of data. Finally, we evaluate the system by using Anvil as the back end for SQLite [51] and as a standalone data store, the latter allowing us to configure it more freely and obtain results similar to recent published works.

3.2 Related Work

In the late 1980s, extensible database systems like Genesis [3] and Starburst [25] explored new types of data layouts, indices, and query optimizers. Starburst in particular defines a "storage method" interface for storage extensibility. This interface features functions for, for example, inserting and deleting a table's rows. Each database table has exactly one storage method and zero or more "attachments," which are used for indexes and table constraints. Anvil's modularity is finer-grained than Starburst's. Anvil implements the functionality of a Starburst storage method through a layered collection of specialized dTables. This increased modularity, and in particular the split between read-only and write-mostly structures, simplifies the construction of new storage methods and method combinations.

Like Starburst, recent versions of MySQL [32] allow users to specify a different storage engine to use for each table. These engines can be loaded dynamically, but again, are not composable. They are also not as easily implemented as Anvil dTables, since they must be read-write while providing the correct transactional semantics. Postgres [52] supported (and now PostgreSQL supports) user-defined index types, but these types cannot control the physical layout of the data itself.

Monet [6] splits a database into a front end and a back end, where the back end has its own query language. While it does not aim to provide the database designer with any modular way of configuring or extending the back end, it does envision that many different front ends should be able to use the same back end.

Stasis [42] is a storage framework providing applications with transactional primitives for an interface very close to that of a disk. Stasis aims for a much lower-level abstraction than Anvil, and expects each application to provide a large part of the eventual storage implementation. Anvil could be built on top of a system like Stasis. This is not necessary, however: Anvil specifically tries to avoid needing strong transactional semantics for most of its data, both for simplicity and to allow asynchronous writes and group commit.

Berkeley DB [36] provides a transactional key-value store, accessed via a library API rather than a network or socket connection, very much like Anvil. It contains several different on-disk data structures among which client applications can choose, and supports concurrent access from many threads as well as "high availability" features like distribution and replication. However, its on-disk layouts cannot be combined or layered together as dTables can; to create a data format combining the properties of several others, an entirely new stand-alone format must be implemented.

Anvil's split between read-only and write-mostly structures relates to read-optimized stores [2, 17, 43] and log-structured file systems [40]. In some sense Anvil carries the idea of a read-optimized store to its limit. Several systems have also investigated batching changes in memory or separate logs, and periodically merging the changes into a larger corpus of data [8, 37, 43]. The functional split between read and write is partially motivated by the increasing discrepancies between CPU speeds, storage bandwidth, and seek times since databases were first developed [18, 54].

We intend Anvil to serve as an experimental platform for specialized stores. Some such stores have reported orders-of-magnitude gains on some benchmarks compared to conventional systems [53]. These gains are obtained using combinations of techniques, including relaxing durability requirements and improving query processing layers as well as changing data stores. We focus only on data stores; the other improvements are complementary to our work. Specifically, Anvil's cTable interface uses ideas and techniques from work on column stores [17, 53],

Bigtable [9], the structured data store for many Google products, influenced Anvil's design. Each Bigtable "tablet" is structured like an Anvil managed dTable configuration: a persistent commit log (like the Anvil transaction library's system journal), an in-memory buffer (like that in the journal dTable), and an overlay of several sorted read-only "SSTables" (like a specific kind of linear dTable). Anvil table creation methods and iterators generalize Bigtable compactions to arbitrary data formats. Anvil's fine-grained modularity helps it support configurations Bigtable does not, such as transparent data transformations and various indices. Bigtable's extensive support for scalability and distribution across large-scale clusters is orthogonal to Anvil, as is its automated replication via the Google File System.

Many of these systems support features that Anvil currently lacks, such as fine-grained locking to support high concurrency, or distribution and replication to support high availability. However, we believe their techniques for implementing these features are complementary to Anvil's modularity.

Anvil aims to broaden and distill ideas from these previous systems, and new ideas, into a toolkit for building data storage layers.

3.3 Design

Two basic goals guided the design of Anvil. First, we want Anvil modules to be fine-grained and easy to write. Implementing behaviors optimized for specific workloads should be a matter of rearranging existing modules (or possibly writing new ones). Second, we want to use storage media effectively by minimizing seeks, instead aiming for large contiguous accesses. Anvil achieves these goals by explicitly separating read-only and write-mostly components, using stacked data storage modules to combine them into read/write stores. Although the Anvil design accommodates monolithic read/write stores, separating these functions makes the individual parts easier to write and easier to extend through module layering. In this section, we describe the design of our data storage modules, which are called dTables.

3.3.1 dTables

dTables implement the key-value store interface summarized in Figure 3.1. For reading, the interface provides both random access by key and seekable, bidirectional iterators that yield elements in sorted order. Some dTables implement this interface directly, storing data in files, while others perform additional bookkeeping or transformations on the data and leave storage up to one or more other dTables stacked underneath.

To implement a new dTable, the user writes a new dTable class and, usually, a new iterator class that understands the dTable's storage format. However, iterators and dTable objects need not be paired: some layered dTables pass through iterator requests to their underlying tables, and some iterators are not associated with any single table.

Many dTables are read-only. This lets stored data be optimized in ways that would be impractical for a writable dTable – for instance, in a tightly packed array with no space for new records, or compressed using context-sensitive algorithms [73]. The creation pro-

```
class dtable {
  bool contains(key_t key) const;
  value_t find(key_t key) const;
  iter iterator() const;
  class iter {
    bool valid() const;
    key_t key() const;
    value_t value() const;
    bool first();          // return true if new position is valid
    bool next();
    bool prev();
    bool seek(key_t key);
    bool reject(value_t * placeholder);  // used at create time, return true on success
  };
  static int create(string file, iter src);
  static dtable open(string file);
  int insert(key_t key, value_t value);
  int remove(key_t key);
};
```

Figure 3.1: Simplified pseudocode for the dTable interface.

cedure for a read-only dTable takes an iterator for the table's intended data. The iterator yields the relevant data in key-sorted order; the creation procedure stores those key-value pairs as appropriate. A read-only dTable implements creation and reading code, leaving the `insert` and `remove` methods unimplemented. In our current dTables, the code split between creation and reading is often about even. Specific examples of read-only dTables are presented in more detail in Section 3.5.

Specialized dTables can refuse to store some kinds of data. For example, the *array* dTable stores fixed-size values in a file as a packed array; this gives fast indexed access, but values with unexpected sizes cannot be stored. Specialized dTables must detect and report attempts to store illegal values. In particular, when a creation procedure's input iterator yields an illegal value, the creation procedure must *reject* the key-value pair by calling the iterator's `reject` method. This explicit rejection gives other Anvil modules an opportunity to handle the unexpected pair, and allows the use of specialized dTables for values that often, but not always, fit some specialized constraints. The rejection notification

```
int array_dtable::create(string file, iter src) {
  wrfile output(file);
  output.append(src.key());  // min key
  while (src.valid()) {
    value_t value = src.value();
    if (value.size() != configured_size && !src.reject(&value))
      return -1;
    output.append(value);
    src.next();
  }
  return 0;
}
```

Figure 3.2: Simplified pseudocode for the array dTable's `create` method. (This minimal version does not, among other things, check that the keys are actually contiguous.)

travels back to the data source along the chain of layered iterators. If a particular layer's iterator knows how to handle a rejected pair, for instance by storing the true pair in a more forgiving dTable, its `reject` function will store the pair, replace the offending value with a placeholder, and return true. (This placeholder can indicate at lookup time when to check the more forgiving dTable for overrides.) If the rejected pair is not handled anywhere, `reject` will return false and the creation operation will fail. We describe the exception dTable, which handles rejection notifications by storing the rejected values in a separate (more generic) dTable, in Section 3.5.5. Figure 3.2 shows pseudocode for the array dTable's `create` method, including its use of `reject`.

Anvil iterators are used mostly at table creation time, which stresses their scanning methods (`key`, `value`, `valid`, and `next`, as well as `reject`). However, external code, such as our SQLite query processing interface, can use iterators as database cursors. The seeking methods (`seek`, `prev`) primarily support this use.

Other dTables are designed mostly to support writing. Writable dTables are usually created empty and populated by writes.

Although arbitrarily complex mechanisms can be built into a single dTable, complex storage systems are better built in Anvil by composing simpler pieces. For instance, rather than building a dTable to directly store U.S. state names and postal abbreviations efficiently (via dictionary lookup) in a file, a dTable can translate state names to dictionary indices and then use a more generic dTable to store the translated data. Likewise, instead of designing an on-disk dTable that keeps a B-tree index of the keys to improve lookup locality, a passthrough B-tree dTable can store, in a separate file, a B-tree index of the keys in another dTable. Further, these two dTables can be composed, to get a B-tree indexed dTable that stores U.S. states efficiently. Similar examples are discussed further in Sections 3.5.2 and 3.5.4.

3.3.2 Data Unification

An Anvil table representation will usually consist of several read-only dTables, created at different times, and one writable dTable. Using this representation directly from client code would inconveniently require consultation of all the dTables. In addition, the periodic conversion of write-optimized dTables to read-only dTables requires careful use of transactions, something that applications should be able to avoid. Anvil includes two key dTables which deal with these chores, combining the operations of arbitrary readable and writable dTables into a single read/write store. We introduce these dTables here; they are discussed in greater detail in Section 3.5.3.

The *overlay* dTable builds the illusion of a single logical dTable from two or more other dTables. It checks a list of subordinate dTable elements, in order, for requested keys, allowing dTables earlier in the list to override values in later ones. This is, in principle, somewhat like the way Unionfs [67] merges multiple file systems, but simpler in an important way: like most dTables, the overlay dTable is read-only. The overlay dTable also merges its subordinates' iterators, exporting a single iterator that traverses the unified data. Significantly, this means that an overlay dTable iterator can be used to create a single new

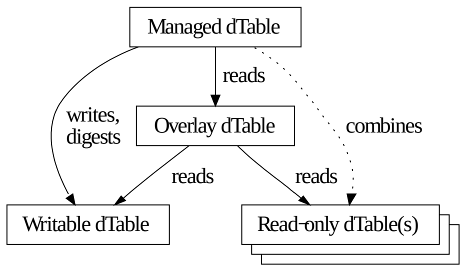


Figure 3.3: The relationships between a managed dTable and the dTables it uses.

read-only dTable that combines the data of its subordinates.

The *managed* dTable automates the use of these overlay dTables to provide the interface of a read/write store. This dTable is an essential part of the typical Anvil configuration (although, for example, a truly read-only data store wouldn't need one). It is often a root module in a dTable module subgraph. Its direct subordinates are one writable dTable, which satisfies write requests, and zero or more read-only dTables, which contain older written data; it also maintains an overlay dTable containing its subordinates. Figure 3.3 shows a managed dTable configuration.

Each managed dTable periodically empties its writable dTable into a new read-only dTable, presumably improving access times. We call this operation *digesting*, or, as the writable dTable we currently use is log-based, *digesting the log*. The managed dTable also can merge multiple read-only dTables together, an operation called *combining*. Without combining, small digest dTables would accumulate over time, slowing the system down and preventing reclamation of the space storing obsoleted data. Combining is similar in principle to the "tuple mover" of C-Store [53], though implemented quite differently. In C-Store, the tuple mover performs bulk loads of new data into read-optimized (yet still writable) data stores, amortizing the cost of writing to read-optimized data structures. In Anvil, however, the managed dTable writes new read-only dTables containing the merged data, afterwards deleting the original source dTables, a process corresponding more closely to Bigtable's merging and major compactions.

```
class ctable {
  bool contains(key_t key) const;
  value_t find(key_t key, int col) const;
  iter iterator(int cols[], int ncols) const;
  int index_of(string name) const;
  string name_of(int index) const;
  int column_count() const;
  class iter {
    bool valid() const;
    key_t key() const;
    value_t value(int col) const;
    bool first();
    bool next();
    bool prev();
    bool seek(key_t key);
  };
  static int create(string file);
  static ctable open(string file);
  int insert(key_t key, int col, value_t value);
  int remove(key_t key);
};
```

Figure 3.4: A simplified, pseudocode version of the cTable interface.

The managed dTable also maintains metadata describing which other dTables it is currently using and in what capacity. Metadata updates are included in atomic transactions when necessary (using the transaction library described in §3.4), largely freeing other dTables from this concern.

3.3.3 Columns

Another Anvil interface, cTable, represents columnated data. It differs from the dTable interface in that it deals with named columns as well as row keys. cTables use dTables as their underlying storage mechanism. Like writable dTables, they are created empty and populated by writes. Figure 3.4 shows a simplified version of the cTable interface.

Anvil contains two primitive cTable types (though like the dTable interface, it is extensible and would support other feature combinations). The first primitive, the row cTable, packs the values for each column together into a single blob, which is stored in a single underlying dTable. This results in a traditional row-based store where all the columns of a row are stored together on disk. The second, the column cTable, uses one underlying dTable per column; these dTables can have independent configurations. A row cTable's iterator is a simple wrapper around its underlying dTable's iterator, while a column cTable's iterator wraps around multiple underlying iterators, one per column.

In a column-based arrangement, it is possible to scan a subset of the columns without reading the others from disk. To support this, cTable iterators provide a *projection* feature, where a subset of the columns may be selected and iterated. A list of relevant column indices is passed to the iterator creation routine; the returned iterator only provides access to those columns. A column cTable's iterator does not iterate over unprojected columns, while a row cTable's iterator ignores the unwanted column data when it is unpacking the blob for each row. We compare the merits of these two cTables in Section 3.7.4.

3.3.4 Discussion

Anvil is implemented in C++, but also provides an API for access from C. All dTable implementations are C++ classes. There is also a dTable iterator base class from which each of the dTables' iterator classes inherit.[1] While many parts of Anvil are currently single-threaded, some dTables support limited use of additional threads, and the relevant core data classes (e.g., file caches and reference-counted blobs) are all thread-safe.

An Anvil instance is provided at startup with a configuration string describing the layout pattern for its dTables and cTables. The initialization process creates objects according to this configuration, which also specifies dTable parameters, such as the value size appropriate for an array dTable. The dTable graph in a running Anvil data store will not

[1]This is a departure from the STL iterator style: iterators for different types of dTables need different runtime implementations, but must share a common supertype.

exactly equal the static configuration, since dTables like the managed dTable can create and destroy subordinates at runtime. However, the configuration does specify what kinds of dTables are created.

dTables that store data on disk do so using files on the underlying file system; each such dTable owns one or more files.

Although our current dTables ensure that iteration in key-sorted order is efficient, this requirement is not entirely fundamental. Iteration over keys is performed only by dTable `create` methods, whereas most other database operations use `lookup` and similar methods. In particular, the dTable abstraction could support a hash table implementation that could not yield values in key-sorted order, as long as that dTable's iterators never made their way to a conventional dTable's `create` method. (See, for instance, the optimization discussed in §3.6.)

Anvil was designed to make disk accesses largely sequential, avoiding seeks and enabling I/O request consolidation. Its performance benefits relative to B-tree-based storage engines come largely from sequential accesses. Although upcoming storage technologies, such as solid-state disks, will eventually reduce the relative performance advantage of sequential requests, Anvil shows that good performance on spinning disks need not harm programmability, and we do not believe a new storage technology would require a full redesign.

Our evaluation shows that the Anvil design performs well on several realistic benchmarks, but in some situations its logging, digesting, and combining mechanisms might not be appropriate no matter how it is configured. For instance, in a very large database that is queried infrequently and regularly overwritten, the work to digest log entries would largely be wasted due to infrequent queries. Further, obsolete data would build up quickly as most records in the database are regularly updated. Although combine operations would remove the obsolete data, scheduling them as frequently as would be necessary would cause even more overhead.

3.4 Transaction Library

Anvil modules use a common *transaction library* to access persistent storage. This library abstracts the file-system-specific mechanisms that keep persistent data both *consistent* and *durable.* Anvil state is *always* kept consistent: if an Anvil database crashes in a fail-stop manner, a restart will recover state representing some prefix of committed transactions, rather than a smorgasbord of committed transactions, uncommitted changes, and corruption. In contrast, users choose when transactions should become durable (committed to stable storage).

The transaction library's design was constrained by Anvil's modularity on the one hand, and by performance requirements on the other. dTables can store persistent data in arbitrary formats, and many dTables with different requirements cooperate to form a configuration. For good performance on spinning disks, however, these dTables must cooperate to group-commit transactions in small numbers of sequential writes. Our solution is to separate consistency and durability concerns through careful use of file-system-specific ordering constraints, and to group-commit changes in a shared log called the *system journal*. Separating consistency and durability gives users control over performance without compromising safety, since the file system mechanisms used for consistency are much faster than the synchronous disk writes required for durability.

3.4.1 Consistency

The transaction library provides consistency and durability for a set of small files explicitly placed in its care. Each transaction uses a file-system-like API to assign new contents to some files. (The old file contents are replaced, making transactions idempotent.) The library ensures that these small files always have consistent contents: after a fail-stop crash and subsequent recovery, the small files' contents will equal those created by some prefix of committed transactions. More is required for full data store consistency, however, since the small library-managed files generally refer to larger files managed elsewhere. For example, a small file might record the commit point in a larger log, or might name the current version of a read-only dTable. The library thus lets users define consistency relationships between other data files and a library-managed transaction. Specifically, users can declare that a transaction must not commit until changes to some data file become persistent. This greatly eases the burden of dealing with transactions for most dTables, since they can enforce consistency relationships for their own arbitrary files.

The library maintains an on-disk log of updates to the small files it manages. API requests to change a file are cached in memory; read requests are answered from this cache. When a transaction commits, the library serializes the transaction's contents to its log, `mdtx.log`. (This is essentially a group commit, since the transaction might contain updates to several small files. The library currently supports at most one uncommitted transaction at a time, although this is not a fundamental limitation.) It then updates a commit record file, `mdtx.cmt`, to indicate the section of `mdtx.log` that just committed. Finally, the library plays out the actual changes to the application's small files. On replay, the library runs through `mdtx.log` up to the point indicated by `mdtx.cmt` and makes the changes indicated.

To achieve consistency, the library must enforce a *dependency ordering* among its writes: `mdtx.log` happens before (or at the same time as) `mdtx.cmt`, which happens before (or at the same time as) playback to the application's small files.

This ordering could be achieved by calls like `fsync`, but such calls achieve durability as well as ordering and are extremely expensive on many stable storage technologies [27]. Anvil instead relies on file-system-specific mechanisms for enforcing orderings. By far

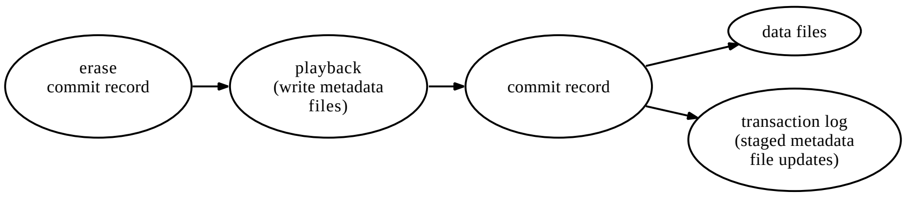


Figure 3.5: Patchgroups used to implement an Anvil transaction. Ovals represent patchgroups, which encompass sets of file system operations. Outgoing arrows represent dependencies on other patchgroups, similar to arrows in patch diagrams.

the simpler of the mechanisms we've implemented is the explicit specification of ordering requirements using the Featherstitch storage system's *patchgroup* abstraction [14]. The transaction library's patchgroups define ordering constraints that the file system implementation must obey. Explicit dependency specification is very clean, and simple inspection of the generated dependencies can help verify correctness. Anvil's ordering requirements in building each transaction translate directly to patchgroups, as shown in Figure 3.5.

Additionally, while running actual crash tests for Anvil's consistency mechanisms (as in §3.7.8) is still a good idea, using patchgroups allows us to decompose a theoretical proof of its correctness into two simpler steps. In one step, we verify that the implementation of patchgroups and of patches in general is correct, as in Section 2.7.3. (This step needs to be done only once, for Featherstitch itself.) In the second step, we verify that the patchgroups Anvil generates are correct by inspection using the Featherstitch patchgroup debugger, which shows the generated patchgroups and their dependencies visually.

Despite these benefits of using patchgroups, we eventually abandoned this approach, for two reasons. First, systems like Featherstitch are not widely deployed, and we wanted to allow Anvil to run on as many systems as possible. Second, Anvil created enough patchgroups that a substantial amount of memory was used to store the structural empty patches used to implement them. We believe that further optimizations to patches, and in particular to empty patches and their use by patchgroups, could alleviate this issue; however, no further work on Featherstitch is planned. (The simpler examples in §2.5.2 generate orders of magnitude fewer patchgroups than Anvil, and thus do not exhibit any problems using patchgroups; in fact, the performance of the IMAP server improves.) Fortunately, ext3's behavior is sufficiently dependable that it can be used to simulate the disk ordering requirements Anvil needs.

Anvil thus currently relies on the accidental [59] write ordering "guarantees" provided by Linux's ext3 file system in ordered data mode. This mode makes two guarantees to applications. First, metadata operations (operations other than writes to a regular file's data blocks) are made in atomic epochs, 5 seconds in length by default. Second, writes to the data blocks of files, including data blocks allocated to extend a file, will be written before the current metadata epoch. In particular, if an application writes to a file and then renames that file (a metadata operation), and the rename is later observed after a crash, then the writes to the file's data blocks are definitely intact.

Anvil's transaction library, like the Subversion [55] working copy library, uses this technique to ensure consistency. Concretely, the `mdtx.cmt` file, which contains the commit record, is written elsewhere and renamed. This rename is the atomic commit point. For example, something like the following system calls would commit a new version of a 16-byte `sysjnl.md` file:

```
pwrite("mdtx.log", [sysjnl.md => new contents], ...)
pwrite("mdtx.cmt.tmp", [commit record], ...)
rename("mdtx.cmt.tmp", "mdtx.cmt") <- COMMIT
pwrite("sysjnl.md.tmp", [new contents], ...)
rename("sysjnl.md.tmp", "sysjnl.md")
```

The last two system calls play out the changes to `sysjnl.md` itself. Writing to `sysjnl.md` directly would not be safe: ext3 might commit those data writes before the `rename` metadata write that commits the transaction. Thus, playback also uses the `rename` technique to ensure ordering. (This property is what makes the transaction library most appropriate for small files.)

The library maintains consistency between other data files and the current transaction using similar techniques. For example, in ext3 ordered data mode, the library ensures that specified data file changes are written before the `rename` commits the transaction.

As an optimization, the transaction library actually maintains only one commit record file, `mdtx.cmt.`*`N`*. Fixed-size commit records are appended to it, and it is renamed so that *`N`* is the number of committed records. Since the transaction library's transactions are small, this allows it to amortize the work of allocating and freeing the inode for the commit record file over many transactions. After many transactions, the file is deleted and recreated.

Much of the implementation of the transaction library is shared between the Featherstitch and ext3 versions, as most of the library's code builds transactions from a generic "write-before" dependency primitive. When running Anvil on Featherstitch, we used dependency inspection tools to verify that the correct dependencies were generated. Although dependencies remain implicit on ext3, the experiments in Section 3.7.8 add confidence that our ext3-based consistency mechanisms are correct in the face of failures.

3.4.2 Durability

As described so far, the transaction library ensures consistency, but not durability: updates to data are not necessarily stored on the disk when a success code is returned to the caller, or even when the Anvil transaction is ended. Updates will eventually be made durable, and many updates made in a transaction will still be made atomic, but it is up to the caller to explicitly flush the Anvil transaction (forcing synchronous disk access) when strong durability is required. For instance, the caller might force durability for network-requested

Class	dTable	Writable?	Description	Section
Storage	Linear	No	Stores arbitrary keys and values in sorted order	3.5.1
	Fixed-size	No	Stores arbitrary keys and fixed-size values in sorted order	3.5.4
	Array	No	Stores consecutive integer keys and fixed-size values in an array	3.5.4
	Indexed	No	Stores consecutive integer keys and arbitrary values in sorted order	3.5.4
	Unique-string	No	Compresses common strings in values	3.5.1
	Empty	No	Read-only empty dTable	3.5.1
	Memory	Yes	Non-persistent dTable	3.5.1
	Journal	Yes	Collects writes in the system journal	3.5.1
Performance	B-tree	No	Speeds up lookups with a B-tree index	3.5.2
	Bloom filter	No	Speeds up nonexistent key lookups with a Bloom filter	3.5.2
	Sliding window	No	Stores spatially local repeated values only once in underlying dTables	3.5.2
	Cache	Yes	Speeds up lookups with an LRU cache	3.5.2
Unifying	Overlay	No	Combines several read-only dTables into a single view	3.5.3
	Managed	Yes	Combines read-only and journal dTables into a read/write store	3.5.3
	Partition	Yes	Partitions key space among underlying dTables	3.5.3
	Exception	No	Reroutes rejected values from a specialized store to a general one	3.5.5
	Existential	No	Stores nonexistent values (§3.5.3) separately from others	3.5.6
Transforming	Small integer	No	Trims integer values to smaller byte counts	3.5.4
	Delta integer	No	Stores the difference between integer values	3.5.4
	State dictionary	No	Maps state abbreviations to small integers	3.5.4
Feature	ACID transaction	Yes	Provides full ACID transaction semantics	3.6

Figure 3.6: Summary of dTables. Storage dTables write data on disk; all other classes layer over other dTables.

transactions only just before reporting success, as is done automatically in the xsyncfs file system [35].

When requested, Anvil makes the most recent transaction durable in one of two ways, depending on whether it is using Featherstitch or ext3. With Featherstitch, it uses the `pg_sync` API to explicitly request that the storage system flush the change corresponding to that transaction to disk. With ext3, Anvil instead calls `futimes` to set the timestamp on an empty file in the same file system as the data, and then `fsync` to force ext3 to end its transaction to commit that change. (Using `fsync` without the timestamp change is not sufficient; the kernel realizes that no metadata has changed and flushes only the data blocks without ending the ext3 transaction.) Even without an explicit request, updates are made durable within about 5 seconds (the default duration of ext3 transactions), as each ext3 transaction will make all completed Anvil transactions durable. This makes Anvil transactions lightweight, since they can be batched and committed as a group.

3.4.3 System Journal

Rather than using the transaction library directly, writable dTables use logging primitives provided by a shared logging facility, the *system journal*. The main purpose of this shared, append-only log is to group writes for speed. Any system component can acquire a unique identifier, called a *tag*, which allows it to write entries to the system journal. Such entries are not erased until their owner explicitly releases the corresponding tag, presumably after the data has been stored elsewhere. Until then, whenever Anvil is started (or on demand), the system journal will replay the log entries to their owners, allowing them to reconstruct their internal state. Appends to the system journal are grouped into transactions using the transaction library, allowing many log entries to be stored quickly and atomically.

To reclaim the space used by released records, Anvil periodically *cleans* the system journal by copying all the live records into a new journal and atomically switching to that version using the transaction library. As an optimization, cleaning is automatically performed whenever the system journal detects that the number of live records reaches zero, since then the file can be deleted without searching it for live records. In our experiments, this actually happens fairly frequently, since entire batches of records are relinquished together during digest operations.

The system journal also supports a "rollover" feature, allowing records originally written using one tag to be sent to the owner of a different tag during playback. This enables several higher-level transaction concepts, like independence and abortability, which are discussed in Section 3.6.

Writing records from many sources to the same system journal is similar to the way log data for many tablets is stored in a single physical log in Bigtable [9]; both systems employ this idea in order to better take advantage of group commit and avoid seeks. Cleaning the system journal is similar to compaction in a log-structured file system, and is also reminiscent of the way block allocation logs (“space maps”) are condensed in ZFS [72].

3.5 dTables

We now describe the currently implemented dTable types and their uses in more detail. The twenty-one types are summarized in Figure 3.6. We close with an example Anvil configuration using many of these dTables together, demonstrating how simple, reusable modules can combine to implement an efficient, specialized data store.

3.5.1 Basic Storage dTables

The dTables described in this section store data directly on disk, rather than layering over other dTables. This section presents only some of Anvil’s storage dTables; several others are described later as uses for them are discussed.

Journal dTable The *journal* dTable is Anvil’s fundamental writable store. The goal of the journal dTable is thus to make writes fast without slowing down reads. Scaling to large stores is explicitly not a goal: large journals should be digested into faster, more compressed, and easier-to-recover forms, namely read-only dTables. Managed dTables in our configurations collect writes in journal dTables, then digest that data into other, read-optimized dTables.

The journal dTable stores its persistent data in the system journal. Creating a new journal dTable is simple: a system journal tag is acquired and stored in a small file managed by the transaction library (probably one belonging to a managed dTable). Erasing a journal dTable requires relinquishing the tag and removing it from the small file. These actions are generally performed at a managed dTable’s request.

Writing data to a journal dTable is accomplished by appending a system journal record with the key and value. However, the system journal stores records in chronological order, whereas a journal dTable must iterate through its entries in sorted key order. This mismatch is handled by keeping an in-memory balanced tree of the entries. When a journal dTable is initialized during Anvil startup, it requests its records from the system journal and replays the previous sequence of inserts, updates, and deletes in order to reconstruct this in-memory state. The memory this tree requires is one reason large journal dTables should be digested into other forms.

Linear dTable The *linear* dTable is Anvil's most basic read-only store. It accepts any types of keys and values without restriction, and stores its data as a simple file containing first a ⟨key, offset⟩ array in key-sorted order, followed by the values in the same order. (Keys are stored separately from values since most of Anvil's key types are fixed-size, and thus can be easily binary searched to allow random access. The offsets point into the value area.) As with other read-only dTables, a linear dTable is created by passing an iterator for some other dTable to a `create` method, which creates a new linear dTable on disk containing the data from the iterator. The linear dTable's `create` method never calls `reject`.

Others The *memory* dTable keeps its data exclusively in memory. When a memory dTable is freed or Anvil is terminated, the data is lost. Like the journal dTable, it is writable and has a maximum size limited by available memory. Our test frameworks frequently use the memory dTable for their iterators: a memory dTable is built up to contain the desired key-value pairs, then its iterator is passed to a read-only dTable's `create` method.

The *empty* dTable is a read-only table that is always empty. It is used whenever a dTable or iterator is required by some API, but the caller does not have any data to provide.

The *unique-string* dTable detects duplicate strings in its data and replaces them with references to a shared table of strings. This approach is similar to many common forms of data compression, though it is somewhat restricted in that it "compresses" each blob individually using a shared dictionary.

3.5.2 Performance dTables

These dTables aim to improve the performance of a single underlying dTable stack by adding indexes or caching results, and begin to demonstrate benefits from layering. For instance, the Bloom filter dTable improves the performance of nonexistent key lookups by a factor of 24 in a simple benchmark, and can easily be added to any read-only dTable stack.

B-tree dTable The *B-tree* dTable creates a B-tree [4] index of the keys stored in an underlying dTable, allowing those keys to be found more quickly than by, for example, a linear dTable's binary search.[2] It stores this index in another file alongside the underlying dTable's data. The B-tree dTable is read-only (and, thus, its underlying dTable must also be read-only). Its `create` method constructs the index; since it is given all the data up front, it can calculate the optimal constant depth for the tree structure and bulk load the resulting tree with keys. This bulk-loading is similar to that used in Rose [43] for a similar purpose, and avoids the update-time complexity usually associated with B-trees (such as rebalancing, splitting, and combining pages).

Bloom Filter dTable The *Bloom filter* dTable's `create` method creates a Bloom filter [5] of the keys stored in an underlying read-only dTable. Like the B-tree dTable, it stores this auxiliary data in a file alongside the underlying dTable's data. It responds to

[2]The asymptotic runtime is the same, but the constant is different: $\log_n x$ instead of $\log_2 x$.

a lookup request by taking a 128-bit hash of the key, and splitting it into a configurable number of indices into a bitmap. If any of the corresponding bits in the bitmap are not set, the key is guaranteed not to exist in the underlying dTable; this result can be returned without invoking the underlying table's lookup algorithm. This is particularly useful for optimizing lookups against small dTables, such as those containing recent changes, that overlay much larger data stores, a situation that often arises in Anvil. The Bloom filter dTable keeps the bitmap cached in memory, as the random accesses to it would not be efficient to read from disk. Like the B-tree dTable, the Bloom filter dTable is able to sidestep some of the usual problems associated with a classic data structure by being read-only: in this case, key removal. Section 3.7.2 evaluates the effectiveness of the Bloom filter dTable.

Sliding Window dTable The *sliding window* dTable eliminates duplicate values by attempting to store only a single copy of each unique value it sees during creation. For data sets with many repeated values, this can dramatically reduce the disk space used and thus the I/O bandwidth required to read such data sets. To do this efficiently, it uses a sliding window approach as seen in the Lempel-Ziv [73] family of data compression algorithms. When a value not currently in the window is seen, it is stored in a subordinate "data" dTable indexed by contiguous integer keys (the indexed or array dTables would be good choices for this). Separately, in a second subordinate "key" dTable, the real key is associated with a fixed-size value that is actually the key to the data dTable where the correct value can be found: either the newly-added value, or a value from earlier in the sliding window. Reading from sliding window dTables is easy: first the desired key is looked up in the key dTable, and the value found there is used as a key to look up the real value. However, it can be expensive to create sliding window dTables, because of the additional work required to perform the value comparisons during compression. Section 3.7.4 evaluates the utility of this sort of compression, and points out some possible changes to Anvil's design that would help dTables like the sliding window dTable.

Cache dTable The *cache* dTable wraps another dTable, caching looked up keys and values so that frequently- and recently-used keys need not be looked up again. If the underlying dTables perform computationally expensive operations to return requested data, such as some kinds of decompression, and some keys are looked up repeatedly, a cache dTable may be able to improve performance. When the underlying dTable supports writing, the cache dTable does as well: it passes the writes through and updates its cache if they succeed. Each cache dTable can be configured with how many keys to store; other policies, such as total size of cached values, would be easy to add.

3.5.3 Unifying dTables

This section describes the overlay and managed dTables in more detail, explaining how they efficiently unify multiple underlying dTables into a seamless-appearing whole. These dTables are of particular importance because they appear in nearly every dTable configuration, playing a central role. It is therefore important that they be implemented as efficiently as possible; their design is one of Anvil's contributions. This section also describes the partition dTable, which is unusual in that it may often appear above a managed dTable in configuration graphs.

Overlay dTable The overlay dTable combines the data in several underlying dTables into a single logical dTable. It does not store any data of its own. An overlay dTable is not itself writable, although writes to underlying dTables will be reflected in the combined data. Thus, overlays must deal with two main types of access: keyed lookup and iteration.

Keyed lookup is straightforward: the overlay dTable just checks the underlying dTables in order until a matching key is found, and returns the associated value. However, a dTable early in the list should be able to "delete" an entry that might be stored in a later dTable. To support this, the `remove` implementation in a writable dTable normally stores an explicit *"nonexistent"* value for the key. These values resemble Bigtable's deletion entries and the whiteout directory entries of Unionfs [67]. Storage dTables are responsible for translating nonexistent values into the appropriate persistent bit patterns, or for rejecting nonexistent values if they cannot be stored. A nonexistent value tells the overlay dTable to skip later dTables and immediately report that the key's value does not exist. Creating a read-only dTable from the writable dTable will copy the nonexistent value just like any other value. When a key-value pair is ignored by an overlay dTable because another value for the key exists earlier in the list, we say that the original key-value pair has been *shadowed*.

Composing an overlay dTable iterator is more difficult to do efficiently. Keys from different underlying iterators must be interleaved together into sorted order, and keys that have been shadowed must be skipped. However, we want to minimize the overhead of doing key comparisons – especially duplicate key comparisons – since they end up being the overlay's primary use of CPU time. The overlay iterator therefore maintains some additional state for each underlying iterator: primarily, whether that iterator points at the current key, a shadowed key, an upcoming key, or a previous key. This information helps the overlay iterator to avoid many duplicate comparisons by partially caching the results of previous comparisons. (A more advanced version might also keep the underlying iterators sorted by the next key each will output.)

Managed dTable The managed dTable plays a central role in most dTable configurations, coordinating the operation of subordinate dTables to hide the complexity of their interactions. It keeps a writable journal dTable to absorb writes, and periodically digests it (creating new read-only dTables) to keep memory usage reasonable and allow space in the system journal to be reclaimed. Less frequently, it also combines several read-only dTables together, improving access times and allowing shadowed, obsolete data to be removed. It must deal with scheduling these operations at reasonable intervals, since performing them too frequently is expensive, yet postponing them too long will degrade overall system performance. Fortunately, digesting and combining can safely be done in the background, and the managed dTable contains special support to make this possible. By handling these coordination tasks, and interfacing with the Anvil transaction library, the managed dTable relieves most other dTables of any need to worry about transactions – and, in fact, consistency in general.

The managed dTable stores its metadata as well as the files for its underlying dTables in a directory. In addition to the journal dTable to which it sends all writes, the managed dTable keeps zero or more other dTables which, along with the journal dTable, are composed using an overlay dTable. A `maintain` method runs scheduled maintenance tasks: digesting the journal dTable to form a new read-only dTable, or combining several dTables to form a single new dTable. (Currently, client code is responsible for periodically calling this method.)

Digesting The current managed dTable schedules digest operations at fixed, configurable intervals. A digest will occur soon after the interval has elapsed, unless the journal dTable is empty. To digest the log, an iterator for the journal dTable is passed to a suitable `create` method to create a new read-only dTable. Once the new dTable is created, the transaction library is used to write new metadata referencing it and the journal dTable's system journal tag is relinquished (an operation that also uses the transaction library); a new system journal tag is also allocated for use by subsequent writes. This operation is similar to system journal cleaning, in that the actual data is written non-transactionally, but the transaction library is used to atomically "swap in" the newly written data.

Combining Combines are more expensive than digests, since they must scan possibly-uncached dTables and create single, larger versions; further, the overlay dTable necessary to merge data from uncombined dTables can be costly as well. Still, combines must be done often enough to keep the number of read-only dTables reasonable: the more there are, the less efficient overlaying them will be and the slower lookups will get. To amortize the cost of combining, combines are scheduled using a "ruler function" somewhat reminiscent of generational garbage collection. A combine operation is performed every k digests ($k = 4$ in our current implementation). The combine takes as input the most recent k digests, plus a number of older dTables according to the sequence $x_i = 0,1,0,2,0,1,0,3,0,1,0,2,\ldots$, where x_i is one less than the number of low-order zero bits in i. This performs small combines much more frequently than large combines, and the average number of dTables grows at most logarithmically with time. The result is very similar to Lester et al.'s "geometric partitioning" mechanism [24]. (A more advanced version of the managed dTable might involve more carefully tuned parameters, or decide when to perform digests and combines based on, for instance, the amount of data involved.) Combining itself is very similar to digesting: an overlay dTable is created to merge the dTables to be combined, and an iterator for that is passed to the appropriate `create` method. Afterward, appropriate metadata is updated and then-unreferenced source dTables marked for deletion.

Obsolete Data When multiple dTables are combined, older data for an updated key is automatically left out since the overlay dTable's iterator will skip it. However, if the key is associated with a nonexistent value, indicating that it has been deleted, it may be possible to remove the key entirely during a combine. To allow `create` methods to determine when this is safe, the managed dTable passes an extra parameter to them: a "shadow dTable" that contains all those keys that might need to be shadowed by the nonexistent values. The

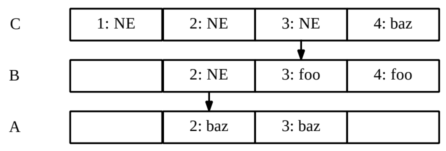


Figure 3.7: dTables C, B, and A in an overlay configuration; explicitly nonexistent values are shown as "NE." Digesting C should keep the nonexistent value for key 3, but not those for keys 1 and 2, since the combination of the other dTables already hold no values for those keys (B takes precedence over A). Combining C and B, on the other hand, must keep the nonexistent values for both keys 2 and 3. The arrows point from required nonexistent values to their shadowed versions.

shadow dTable is simply an overlay dTable that merges all the dTables that are not being combined, but that the newly created dTable might shadow. The `create` method can then look up keys with nonexistent values in the shadow dTable to see if the explicit nonexistent values are still required. Figure 3.7 shows an example set of dTables to help illustrate this algorithm.

Background Operation A managed dTable's digesting and combining can be safely done in a background thread, keeping these potentially lengthy tasks from blocking other processing. As a combine operation reads from read-only dTables and creates a new dTable that is not yet referenced, the only synchronization required is at the end of the combine when the newly-created dTable replaces the source dTables. Digests read from writable journal dTables, but can also be done in the background by creating a new journal dTable and marking the original journal dTable as "effectively" read-only before starting the background digest. Such read-only journal dTables are treated by a managed dTable as though they were not journal dTables at all, but rather one of the other read-only dTables.

The current managed dTable implementation allows only one digest or combine operation to be active at any time, whether it is being done in the background or not. Background digests and combines could also be done in a separate process, rather than a separate thread; doing this would avoid the performance overhead incurred by thread safety in the

C library. Section 3.7.5 evaluates the costs associated with digesting and combining, and quantifies the overhead imposed by using threads.

Partition dTable The *partition* dTable, like the overlay dTable, joins several underlying dTables together to appear as a single dTable. However, rather than storing the same keys in each subordinate dTable and overlaying the values, it divides the key space into configured regions and stores only part of the data set in each subordinate dTable. Its iterators simply concatenate the data from subordinate dTables iterators, rather than interleaving it based on keys like overlay dTable iterators. As a writable dTable, it is created empty, and populated later; it sends writes to the appropriate subordinate dTables, which are also expected to be writable. It is one of only three dTables (the others being the cache dTable and ACID transaction dTable) that might normally appear above a managed dTable in a configuration graph, as it can be handy to split a large data set into smaller pieces that can be independently digested or combined – allowing greater parallelization of these tasks. (This parallelization opportunity builds on multiple managed dTables' background digesting: a partition dTable, called from a single thread, can start several background digests without any special code to do so.) Section 3.7.2 evaluates the overhead and benefits provided by the partition dTable.

3.5.4 Specialized dTables

Although a linear dTable can store any keys and values, keys and values that obey some constraints can often be stored in more efficient ways. For instance, if all the values are the same size, then file offsets for values can be calculated based on the indices of the keys and need not be stored. Alternately, if the keys are integers and are likely to be consecutive, the binary search can be optimized to a constant time lookup by using the keys as indices and leaving "holes" in the file where there is no data. Or, if the values are likely to compress well with a specific compression algorithm, that algorithm can be applied. Using techniques like these, when appropriate, can significantly reduce file size, runtime performance, or both.

This section describes several of Anvil's specialized dTables that efficiently store specific kinds of data. The array dTable, fixed-size dTable, and indexed dTable are storage dTables (they store data on disk themselves); the rest layer over other dTables and transform the data. This latter class of specialized dTable can be particularly potent and is one of the big advantages of a modular design like Anvil's: several such layered dTables can be combined and added on top of any underlying dTables, making it easy to build very specialized dTable configurations from more general pieces.

Array dTable The *array* dTable is specialized for storing fixed-size values associated with contiguous (or mostly contiguous) integer keys. After a short header, which contains the initial key and the value size, an array dTable file contains a simple array of values. Each value is optionally preceded by a tag byte to indicate whether the following bytes are actually a value or merely a hole to allow the later values to be in the right place despite the missing key. Without the tag byte, specific values must be designated as the ones to be used to represent nonexistent values and holes, or they will not be supported. (And, in *their* absence, the array dTable's `create` method will fail if a nonexistent value or non-contiguous key is encountered, respectively.)

Fixed-size dTable The *fixed-size* dTable is like the array dTable in that it can only store values of a fixed size. However, it accepts all Anvil-supported key types, and does not require that they be contiguous. While the direct indexing of the array dTable is lost, the size advantage of not saving the value size with every entry is retained.

Indexed dTable Like the fixed-size dTable, the *indexed* dTable is more flexible than the array dTable, but in the other dimension: it can only store contiguous integer keys, but it can store any size values. Like the array dTable it does not need to search to find keys, but like the linear dTable the key array contains only pointers to the values. Each value lookup thus takes two reads, rather than one, but values are free to take on any size as a result.

Small Integer dTable The *small integer* dTable is designed for values that are small integers. It requires that all its input values be 4 bytes (32 bits), and interprets each as an integer in the native endianness. It trims each integer to a configured number of bytes (one of 1, 2, or 3), rejecting values that do not fit in that size, and stores the resulting converted values in another dTable.

Delta Integer dTable Like the small integer dTable, the *delta integer* dTable works only with 4-byte values interpreted as integers. Instead of storing the actual values, it computes the difference between each value and the next and passes these differences to a dTable below. If the values do not usually differ significantly from adjacent values, the differences will generally be small integers – perfect for then being stored using a small integer dTable.

Storing the differences, however, causes problems for seeking to random keys. The entire table, from the beginning, would have to be consulted in order to reconstruct the appropriate value. To address this problem, the delta integer dTable also keeps a separate "landmark" dTable, which stores the original values for a configurable fraction of the keys. To seek to a random key, the landmark dTable is consulted, finding the closest landmark value. The delta dTable is then used to reconstruct the requested value starting from the landmark key.

State Dictionary dTable *Dictionary* dTables compress data by transforming user-friendly values into less-friendly values that require fewer bits. As a toy example, Anvil's *state dic-*

tionary dTable translates U.S. state postal codes (CA, MA, etc.) to and from one-byte numbers. During creation, it translates input postal codes into bytes and passes them to another dTable for storage; during reading, it translates values returned from that dTable into postal codes. The array or fixed-size dTables are ideally suited as subordinates to the state dictionary dTable, especially (in the case of the array dTable) if we use some of the remaining, unused values for the byte to represent holes and nonexistent values.

3.5.5 Exception dTable

The *exception* dTable takes advantage of Anvil's modularity and iterator rejection to efficiently store data in specialized dTables without losing the ability to store any value. This can improve the performance or storage space requirements for tables whose common case values fit some constraint, such as a fixed size.

Like an overlay dTable, an exception dTable does not store any data of its own; it merely combines data from two subordinate dTables into a single logical unit. These are a general-purpose dTable, such as a linear dTable, which stores exceptional values, and a specialized dTable, such as an array dTable, which is expected to hold the majority of the dTable's data. On lookup, the exception dTable checks the special-purpose dTable first. If there is no value there, it assumes that there is also no exception, and need not check. (It ensures that this will be the case in its `create` method.) If there is a value, and it matches a configurable "exception value," then it checks the general-purpose dTable. If a value is found there, then it is used instead.

The exception dTable's `create` method wraps the source iterator in an iterator of its own, adding a `reject` method that collects rejected values in a temporary memory dTable. This wrapped iterator is passed to the specialized dTable's `create` method. When the specialized dTable rejects a value, the wrapped iterator stores the exception and returns the configurable exception value to the specialized dTable, which stores that instead. The general-purpose dTable is later created by digesting the temporary memory dTable.

3.5.6 Existential dTable

Like the exception dTable, the *existential* dTable uses two subordinate dTables to store data, keeping some values in one and other values in the other. However, instead of using the rejection mechanism to determine which values to store in each, it has a built-in policy: it stores explicitly nonexistent values in one dTable, and all other values in the other. This can be useful to keep nonexistent values out of specialized data stores that, while capable of storing them, would store them inefficiently. (The fixed dTable, for instance, is such a dTable.)

The existential dTable is unique in that once created, its job is entirely done by other dTables: at runtime, its factory does not instantiate a specific existential dTable class. Rather, the underlying dTables are instantiated, and a regular overlay dTable is used to combine them. The existential dTable's `create` method is what does most of the work, using wrapper iterators to filter the source iterator while passing it to the subordinate dTables' `create` methods.

3.5.7 Example Configurations

To show how one might build an appropriate configuration for a specific use case, we work through two simple examples that demonstrate Anvil's modularity and configurability. In each example, we will first build a simple dTable configuration to store the data. We will then consider potential issues with it, like performance problems under certain workloads. Finally, we will examine ways in which the configuration can be improved to address the problems, and build a revised dTable configuration.

First, suppose we want to store the (U.S.) states of residence of customers for a large company. The customers have mostly sequential numeric IDs, and occasionally move between states. We start with a managed dTable, since nearly every Anvil configuration needs one to handle writes. This automatically brings along a journal dTable and overlay dTable, but we must configure it with a read-only dTable. Since there are many customers, but they only occasionally move, we are likely to end up with a very large data set but several smaller read-only "patches" to it (the results of digested journal dTables). Since most keys looked up will not be in the small dTables, we add a Bloom filter dTable to optimize nonexistent lookups. Underneath the Bloom filter dTable, we use a B-tree dTable to speed up successful lookups, reducing the number of pages read in from disk to find each record. To finish our first attempt at a configuration for this scenario, we use a linear dTable under the B-tree dTable. This configuration is shown in Figure 3.8.

Considering only the way the data is arranged on disk by this configuration, we might expect to get a data store not unlike this using a more traditional database back end. However, even with the B-tree index, this configuration will spend a lot of time locating keys during lookups. This is a shame, since aside from the occasional international customer, the data is fixed-size and could be stored much more efficiently in an array using the mostly-sequential customer IDs as the index. To make this possible, we can use the state dictionary dTable combined with an exception dTable for international customers. We can then use an array dTable under the state dictionary dTable, since the state codes it will be asked to store will always be a fixed size. (And there will be enough unused values for us to configure the array dTable to use two of them to store holes and nonexistent values.) We could place these dTables under the B-tree dTable, but since a B-tree index of an array is unnecessary, we instead insert the exception dTable directly under the Bloom filter dTable. Finally, we use the original B-tree dTable subgraph as the generic dTable for the exception dTable, used to store the international customer data.

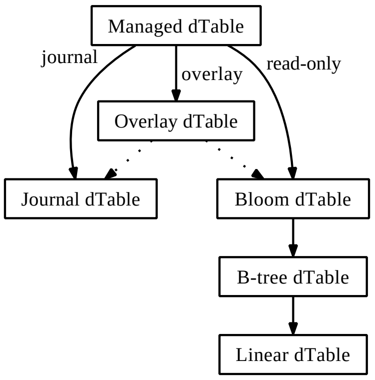


Figure 3.8: A simple dTable graph for the customer state example.

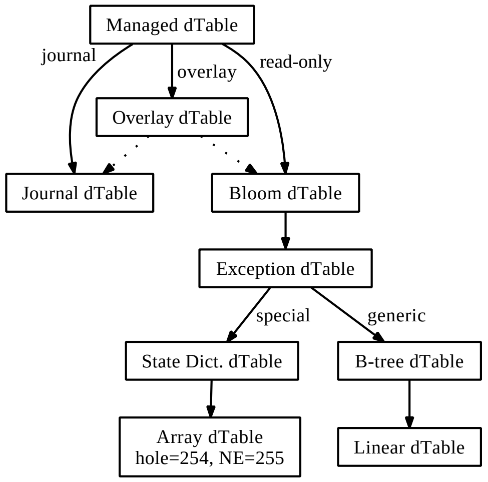


Figure 3.9: An example dTable graph for storing U.S. states efficiently, while still allowing other locations to be stored.

This configuration is shown in Figure 3.9, although at runtime, the managed dTable might create several Bloom filter dTable instances, each of which would then have a copy of the subgraph below.

In a different scenario, this configuration might be just a single column in a column-based store. To see how such a configuration might look, we work through a second example. Suppose that in addition to updating the "current state" table above, we wish to store a log entry whenever a customer moves. Each log entry will be identified by a monotonically increasing log ID, and consist of the pair $\langle$*timestamp*, *customer ID*$\rangle$. Additionally, customers do not move at a uniform rate throughout the year – moves are clustered at specific times of the year, with relatively few at other times.

We start with a column cTable, since we will want to use different dTable configurations for the columns. For the second column, we can use a simple configuration consisting of a managed dTable and array dTables, since the customer IDs are fixed-size and the log IDs are consecutive.

The first column is more interesting. A well-known technique for storing timestamps efficiently is to store the differences between consecutive timestamps, since they will often be small. We therefore begin with a managed dTable using delta integer dTables. The delta integer dTable needs a landmark dTable, as mentioned in Section 3.5.4; we use a fixed dTable as the values will all be the same size. But merely taking the difference in this case is not useful unless we also store the differences with a smaller amount of space than the full timestamps, so we connect the delta integer dTable to a small integer dTable. Finally, we use an array dTable under the small integer dTable to store the consecutively-keyed small integers.

This initial configuration works well during the times of year when many customers are moving, since the differences in timestamps will be small. However, during the other times of the year, when the differences are large, the delta integer dTable will produce large deltas that the small integer dTable will refuse to store. To fix this problem, we need an exception dTable *between* the delta integer dTable and the small integer dTable. Finally, we can use a fixed dTable to store the exceptional values – that is, the large deltas – as they are all the same size. The revised configuration, complete with the configuration for the second column and the containing column cTable, is shown in Figure 3.10.

3.6 Abortable and ACID Transactions

The basic transactions described in Section 3.4 provide atomicity and durability (the A and D in ACID), but cannot provide the other properties. Although the C stands for "consistency" – which as we have defined it the transaction library already provides – in this context it has a different meaning and refers to preservation of database-specific constraints, requiring that transactions that would violate such constraints be aborted. To change the pH of Anvil's basic transactions to make them more acidic, we need to be able to have multiple, isolated transactions in progress simultaneously, and to be able to abort them. Despite

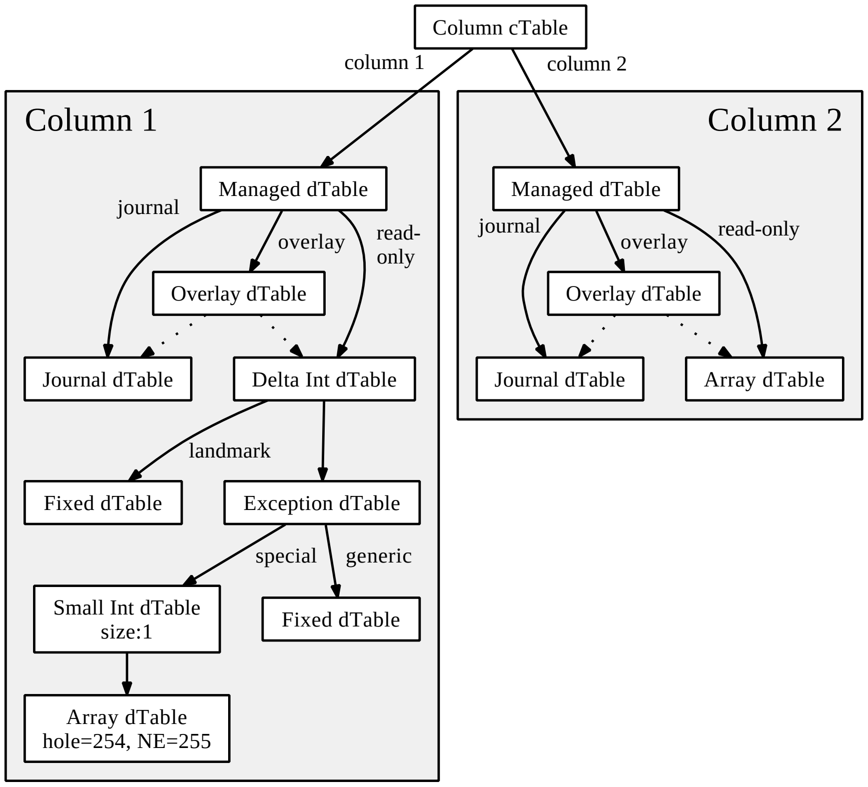


Figure 3.10: An example configuration for a cTable storing differentially timestamped log entries consisting of fixed-size customer IDs.

being one of the later features implemented in Anvil, abortable transactions fit right in – in fact, the modular design made it a very easy addition. In some sense, abortable transactions are themselves a modular feature, implemented within the managed dTable but supported by lower-level functionality.

Each abortable transaction creates its own private journal dTable in which to store its changes, and uses a private overlay dTables to layer them over the “official” stores. The overhead to create these dTables is small, as creating a new journal dTable is a fast operation: it has no files on disk, and merely involves incrementing an integer and allocating object memory. (A simple microbenchmark that creates journal and overlay dTables and then destroys them can create about 600,000 such pairs per second on our benchmark machine.) During the transaction, changes are written to the system journal, just as normal changes that are not in an abortable, independent transaction, but with a different system journal tag. To abort such a transaction, the temporary journal dTable is discarded, and the overlay dTable destroyed. To commit it, the system journal's rollover feature is used. A small record is written to the system journal, listing the system journal tag for the temporary journal dTable and the tag of the current "main" journal dTable. Then the in-memory balanced trees of the two journal dTables are merged, with the abortable transaction's data taking precedence. The transaction's data can then be digested as usual later on, as though it had always been part of the main journal dTable. If the system is shut down (cleanly or otherwise) before the system journal is filtered, these actions will be replayed: a temporary journal dTable will be created for the transaction, populated, and merged into the main journal dTable.

These abortable transactions provide semantics like full ACID transactions that never contain any reads. Although Anvil's abortable transactions *can* be aborted, nothing ever *compels* them to do so. The order in which they commit determines what data will exist in the resulting data store. However, Anvil's modular design allows the read/write tracking required to detect colliding transactions to be done in a separate dTable, making it possible to implement full ACID transactions as a separate dTable. The ACID transaction dTable, evaluated in Section 3.7.7, does exactly that, forcing abortable transactions to abort when it detects read-write or write-write collisions. It can be added at the top of any dTable configuration to provide full read/write ACID transactions, but can be left off when the additional cost to detect collisions is not justified.

As mentioned above, abortable transactions were surprisingly easy to add, and took significant advantage of Anvil's modular design. The rollover feature had to be added to the system journal and journal dTables to support it, but beyond that the changes were very small: about 160 lines of code in the managed dTable. That abortable transactions fit so cleanly and easily into the modular design provides additional evidence that it may be a good design; it also resonates with the overall hypothesis that new features can be added to modular data storage systems with little incremental effort. Using these abortable (or full ACID) transactions is not without cost, however. Compared to using only basic transactions, abortable transactions must allocate two additional dTable objects in memory, write one additional record to the system journal, and, most significantly, merge the journal dTable balanced trees to commit the transaction. ACID transactions must additionally detect colliding reads and writes.

Anvil implements one important optimization, however: the journal dTables used to store the data local to each transaction do not actually keep a balanced tree for the keys unless an iterator is requested by the application. Rather, they keep a hash map of the keys, which is faster, and only convert it to a balanced tree if in-order access to the keys is required. (Normal journal dTables will always be eventually asked to traverse the keys in order during a digest operation, but transaction-local journal dTables will be rolled over – not digested.) This avoids duplicating the work of inserting each key into a balanced tree unless the application requires an iterator inside the transaction. The performance of Anvil's abortable transactions, including this optimization, is evaluated in Section 3.7.6.

3.6.1 Concurrent Access

Along with background digestion, which required the addition of synchronization code in lower-level parts of Anvil to support concurrent access to core data classes, Anvil's abortable transactions also provide most of the mechanisms that would be required to support concurrent access by multiple threads. The major challenges in implementing the remaining pieces would seem to be shared with any system supporting concurrent transactions, namely detecting conflicts and adding locks. Unfortunately, some concurrency disciplines seem perhaps difficult to add as separate modules; for example, every writable dTable might require changes to support fine-grained record locking.

3.7 Evaluation

Anvil decomposes a back-end storage layer for structured data into many fine-grained modules which are easy to implement and combine. Our performance hypothesis is that this modularity comes at low cost for "conventional" workloads, and that simple configuration changes targeting specific types of data can provide significant performance or storage size improvements. This section evaluates Anvil to test our hypothesis and provides experimental evidence that Anvil provides the consistency and durability guarantees we expect.

All tests were run on an HP Pavilion Elite D5000T with a quad-core 2.66 GHz Core 2 CPU, 8 GiB of RAM, and a Seagate ST3320620AS 320 GB 7200 RPM SATA2 disk attached to a SiI 3132 PCI-E SATA controller. Tests use a 24 GiB ext3 file system (and the ext3 version of the transaction library) and the Linux 2.6.24 kernel with the Ubuntu v8.04 distribution. All timing results are the mean over eight runs.

3.7.1 Conventional Workload

For our conventional workload, we use the DBT2 [11] test suite, which is a "fair usage implementation"[3] of the TPC-C [56] benchmark. In all of our tests, DBT2 is configured to run with one warehouse for 15 minutes; this is long enough to allow Anvil to do many digests, combines, and system journal cleanings so that their effect on performance will be measured. We also disable the 1% random rollbacks that are part of the standard bench-

[3]This is a legal term. See the DBT2 and TPC websites for details.

mark, as SQLite's transaction abort mechanism does not easily lend itself to implementation with Anvil's abortable transactions. [4] We modified SQLite, a widely-used embedded SQL implementation known for its generally good performance, to use Anvil instead of its original B-tree-based storage layer. We use a simple dTable configuration: a linear dTable, layered under a B-tree dTable (for combines but not digests), layered under the typical managed and journal dTable combination.

We compare the results to unmodified SQLite, configured to disable its rollback journal, increase its cache size to 128 MiB, and use only a single lock during the lifetime of the connection. We run unmodified SQLite in three different synchronicity modes: *full*, which is fully durable (the default); *normal*, which has "a very small (though non-zero) chance that a power failure at just the wrong time could corrupt the database"; and *async*, which makes no durability guarantees. We also compare to MySQL using two of its "storage engines:" *innodb*, which is fully durable; and *MyISAM*, which makes no durability guarantees. We run SQLite with Anvil in two different modes: *fsync*, which matches the durability guarantee of the original SQLite *full* mode by calling `fsync` at the end of every transaction, and *delay*, which allows larger group commits as described in Section 3.4.2. Both of these modes, as well as the first two unmodified SQLite modes and MySQL's innodb engine, provide consistency in the event of a crash; SQLite's async mode and MySQL's MyISAM engine do not.

The DBT2 test suite issues a balance of read and write queries typical to the "order-entry environment of a wholesale supplier," and thus helps demonstrate the effectiveness of using both read- and write-optimized structures in Anvil. In particular, the system journal allows Anvil to write more data per second than the original back end without saturating

[4]SQLite supports only one outstanding transaction at a time, and makes many assumptions based on this fact that are difficult to satisfy with Anvil's more flexible abortable transactions. For instance, it assumes that all extant iterators automatically "enter" each transaction as soon as it starts, and that they will continue to work after it commits.

	TPM	Disk util.	Avg. reqsz (KiB)	Writes/s
Original, full	905	94.5%	8.52	437.2
Original, normal	920	93.2%	8.69	449.1
Original, async	3469	84.7%	8.73	332.4
MySQL, innodb	1280	84.1%	30.43	640.4
MySQL, MyISAM	4010	74.2%	9.46	445.4
Anvil, fsync	4875	29.6%	24.77	1032.4
Anvil, delay	7835	2.4%	346.71	9.9

Figure 3.11: Results from running the DBT2 test suite. *TPM* represents "new order Transactions Per Minute"; larger numbers are better. *Disk util.* is disk utilization, *Avg. reqsz* the average size in KiB of the issued requests, and *Writes/s* the number of write requests issued per second. I/O statistics come from the `iostat` utility and are averaged over samples taken every minute.

the disk, because its writes are more contiguous and do not require as much seeking. (Anvil actually writes much less in delay mode, however: the average request size increases by more than an order of magnitude, but the number of writes per second decreases by two orders.) For this test, Anvil handily outperforms SQLite's default storage engine while providing the same durability and consistency semantics. Additionally, SQLite using Anvil as its back end – *even in fsync mode* – outperforms MySQL using *either* of its storage engines. The performance advantage of read- and write-optimized structures far outweighs any cost of separating these functions into separate modules.

3.7.2 Microbenchmarks

We further evaluate the performance consequences of Anvil's modularity by stress-testing Anvil's most characteristic modules, namely those dTables that layer above other storage modules.

Exception and Specialized dTables To determine the cost and benefit associated with the exception dTable, we run a model workload with several different dTable configurations and compare the results. For our workload, we first populate a managed dTable with

	Digest (s)	Lookup (s)	Size (MiB)
linear	2.19	66.14	49.6
btree	2.68	24.76	80.2
array	1.77	9.12	22.9
excep+array	1.84	9.39	23.0
excep+fixed	1.89	60.06	34.4
excep+btree+fixed	2.40	17.22	65.0

Figure 3.12: Exception dTable microbenchmark. A specialized array dTable outperforms the general-purpose linear dTable, even if the latter is augmented with a B-tree index. When most, but not all, data fits the specialized dTable's constraints, the exception dTable achieves within 3% of the specialized version while supporting any value type.

4 million values, a randomly selected 0.2% of which are 7 bytes in size and the rest 5 bytes. We then digest the log, measuring the time it takes to generate the read-only dTable. Next we time how long it takes to look up 2 million random keys. Finally, we check the total size of the resulting data files on disk.

We run this test with several read-only dTable configurations. The *linear* configuration uses only a linear dTable. The *btree* configuration adds a B-tree dTable to this. The *array* configuration uses an array dTable instead, and, unlike the other configurations, all values are 5 bytes. The remaining configurations use an exception dTable configured to use a linear dTable as the generic dTable. The *excep+array* configuration uses a 5-byte array dTable as the specialized dTable; the *excep+fixed* configuration uses a 5-byte fixed dTable. Finally, the *excep+btree+fixed* configuration uses a B-tree dTable over a fixed dTable. The results are shown in Figure 3.12.

Comparing the *linear* and *btree* configurations shows that a B-tree index dramatically improves random read performance, at the cost of increased size on disk. For this example, where the data is only slightly larger than the keys, the increase is substantial; with larger data, it would be smaller in comparison. The *array* configuration, in comparison, offers a major improvement in both speed and disk usage, since it can locate keys directly, without search. The *excep+array* configuration degrades *array*'s lookup performance by

	Lookup	Scan
direct	141.2 s	15.63 s
overlay	149.1 s	17.04 s
overhead	5.59%	9.02%

Figure 3.13: Overlay dTable microbenchmark: looking up random keys and scanning tables with and without an overlay. Linear scan overhead is larger percentagewise; a linear scan's sequential disk accesses are so much faster that the benchmark is more sensitive to CPU usage.

only approximately 3% for these tests, while allowing the combination to store any data value indiscriminately. Thus, Anvil's modularity here offers substantial benefit at low cost. The *excep+fixed* configurations are slower by comparison on this benchmark – the fixed dTable must locate keys by binary search – but could offer substantial disk space savings over array dTables if the key space was more sparsely populated.

Overlay dTable All managed dTable reads and combines go through an overlay dTable, making it performance sensitive. To measure its overhead, we populate a managed dTable with 4 million values using the *excep+array* configuration. We digest the log, then insert one final key so that the journal dTable will not be empty. We time how long it takes to look up 32 million random keys, as well as how long it takes to run an iterator back and forth over the whole dTable four times. (Note that the same number of records will be read in each case.) Finally, we open the digested exception dTable within the managed dTable directly, thus bypassing the overlay dTable, and time the same actions. (As a result, the single key we added to the managed dTable after digesting the log will be missing for these runs.)

The results are shown in Figure 3.13. While the overhead of the linear scans is less than that of the random keys, it is actually a larger percentage: the disk accesses are largely sequential (and thus fast) so the CPU overhead is more significant in comparison. As in the last test, the data here is very small; as the data per key becomes larger, the CPU time will

	Populate (s)	Digest (s)	Lookup (s)	Scan (s)
single	15.2	3.8	34.9	6.8
partition	14.2	2.1	31.7	7.7

Figure 3.14: Partition dTable microbenchmark: various tasks with and without a partition dTable.

be a smaller percentage of total time. Nevertheless, this is an important area where Anvil stands to improve. Since profiling indicates key comparison remains expensive, the linear access overhead, in particular, might be reduced by storing precomputed key comparisons in the overlay dTable's iterator, rather than recalculating them each time `next` is called.

Partition dTable Like the overlay dTable, the partition dTable unifies underlying dTables at the cost of adding an additional layer of indirection. To measure the cost of that layer, we compare two dTable configurations: one using a partition dTable with four partitions, each being a managed dTable with subordinate linear dTables, and one using a single managed dTable directly. We populate each with 4 million values between 75 and 85 bytes, then digest the log; when using the partition dTable, the digest proceeds in parallel for each partition. We then look up 1 million random keys and scan an iterator forward over the data three times.

The results, shown in Figure 3.14, show that the overhead is in fact often less than the benefit of using multiple underlying dTables. The random lookup performance and digest performance both show substantial improvement; in the case of the random lookups, this is primarily due to more caching as a side effect of having more dTables. (Linear dTables each contain a file cache. Decreasing the size of each cache so the total size is the same as in the unpartitioned case makes the performance match.) The improvement in digest time, however, is due to the parallelization made possible by the partition dTable. A similar test forcing the digest operations to be executed serially takes as long as the unpartitioned version.

	Even keys (s)	Odd keys (s)	Mixed keys (s)
direct	31.88	26.04	29.15
bloom	33.47	1.09	17.30

Figure 3.15: Bloom filter dTable microbenchmark. A Bloom filter dTable markedly improves lookup times for nonexistent (odd) keys while adding only a small overhead for keys that do exist.

Bloom Filter dTable To evaluate the Bloom filter dTable's effectiveness and cost, we set up an integer-keyed linear dTable with values for every even key in the range 0 to 8 million. (We configure the Bloom filter dTable's hash to produce five 25-bit-long indices into a 4 MiB bitmap.) We then look up 1 million random even keys, followed by 1 million random odd keys, either using a Bloom filter dTable or by accessing the linear dTable directly. The results are shown in Figure 3.15. The Bloom filter dTable adds about 5% overhead when looking up existing keys, but increases the speed of looking up nonexistent keys by nearly a factor of 24. For workloads consisting of many queries for nonexistent keys, this is definitely a major benefit, and the modular dTable design allows it to be used nearly anywhere in an Anvil configuration.

To summarize the microbenchmarks, Anvil's layered dTables add from 3% to 10% overhead for lookups. However, their functionality can improve performance by up to 24 times for some workloads. The combination of small, but significant, overhead and occasional dramatic benefit argues well for a modular design.

3.7.3 Nonexistent Values

From the point of view of storage dTables like the linear dTable, Anvil's explicit nonexistent values are actually quite similar to normal ("extant") values. They key must be stored (if applicable), and mapped to the nonexistent value. In some cases, like the fixed and array dTables, there is no choice but to allocate just as much space as any extant value to store these values. In layered performance dTables, like the B-tree and Bloom filter dTables, keys with nonexistent values must be treated just like keys with extant values. Since this could potentially present significant overhead, we measure a relatively delete-heavy workload to determine how well the system handles this difficult input.

Due to the different ways that different dTables store and process data, we expect the overhead associated with nonexistent values to differ depending on the dTables involved. Some, like the Bloom filter dTable, we expect to be highly affected, as they must treat nonexistent values exactly the same as extant values. Others, like the linear dTable, we expect to be less affected.

To test this, we use two related workloads: first, a 50/50 mix of inserts and deletes to random integer keys in the range $[0, 4000000)$, 12 million operations total (the "reference" workload). Second, the exact same sequence of operations, but skipping any operations (both insert and delete) of keys that, in the reference dTable, did not exist at the end of the test – that is, they had been created and then deleted (the "oracle" workload). We run each workload in two different configurations: one using a Bloom filter dTable and linear dTable together, and one with only the linear dTable. (All configurations use a managed dTable and overlay dTable.) During the workloads we periodically run digests, forcing data to become read-only so that (in the case of the reference workloads) nonexistent values will be necessary. It is not the performance of the workloads themselves that we are interested in measuring, however. The oracle workloads are "cheating" a little too much for that to be a worthwhile comparison: not only do they not have the explicit nonexistent values that we seek to avoid, but they never inserted the original values either. Instead, these workloads are meant to create data stores that differ only in their internal structure, but which contain identical logical data.

On each of these data stores, we then look up 12 million random keys. We also test the effectiveness of the Bloom filters, by calculating the percentage of nonexistent keys –

	Lookup (s)	Bloom Filter Effectiveness	Size (MiB)
reference, Bloom filter	199.8	89.7%	75.2
oracle, Bloom filter	141.5	100.0%	58.5
reference, linear only	933.0	N/A	51.1
oracle, linear only	1011.2	N/A	34.5

Figure 3.16: Effect of nonexistent values on lookup performance and data size after a delete-heavy workload. The oracle Bloom filter effectiveness is actually slightly less than 100%, but with rounding is listed as 100%.

both keys that don't exist at all and those that have explicit nonexistent values – that the Bloom filters can successfully detect. (These are approximately the ones that don't exist at all, save a very small handful of genuine hash collisions in the Bloom filters.) The results are shown in Figure 3.16.

The reference workload generates about 1.9 million extant values, 878,000 nonexistent values, and 1.2 million keys that are never set. (Recall that we are only working with 4 million keys total.) Even with such a sizeable fraction of nonexistent values, the effectiveness of the Bloom filters remains at nearly 90%, and the lookups take only 41% longer. Normally, we would not expect workloads to consist of 50% deletes, nor would we force the managed dTable to digest but not combine – combines being the mechanism by which nonexistent values can be merged with the values they cover, and removed. Under a less-challenging workload, the effect of nonexistent values would be less significant.

The Bloom filter dTable exhibits relatively worst-case performance with respect to nonexistent values, since from its point of view, a key with a nonexistent value is the same as a key with an extant value. Perhaps more interesting, however, is that without the Bloom filter dTable, the linear dTable configuration actually does *better* under the reference workload, *with* nonexistent values. In this configuration, keys that are never set and keys with nonexistent values must both be looked up in linear dTables – while in the Bloom filter configuration, the Bloom filter can nearly always reject keys that are never set without consulting a linear dTable. The linear dTable performs so similarly when looking up these two kinds of keys that a side effect of nonexistent values on overlay dTables actually improves performance by more than the overhead: a nonexistent value is *authoritative*, and when it encounters one, an overlay dTable stops looking for the key and returns without checking any older dTables. The oracle data store, on the other hand, forces the overlay to always check all dTables for these keys, since it does not contain any nonexistent values to stop the lookups early. (Incidentally, it seems entirely plausible that this effect would occur in real workloads, under the assumption that recently-deleted keys are the most likely kind of nonexistent keys to be looked up.)

3.7.4 Reconfiguring Anvil

Many of Anvil's dTable modules do their work on single columns at a time, so they can best be used when Anvil is configured as a column-based store. Other recent work proposing column stores has turned to TPC-H [57], or variants of it, to show the advantages of the approach. However, being a back-end data store and not a DBMS in its own right, Anvil provides no structured query language. Although we have connected it to SQLite to run the TPC-C benchmark, SQLite is a thoroughly row-based system. Thus, in order to demonstrate how a column-based Anvil configuration can be optimized for working with particular data, we must build our own TPC-H-like benchmark, as in previous work [17, 43]. We adapt the method of Harizopoulos et al. [17], as it does not require building a query language or relational algebra.

We create the `lineitem` table from TPC-H, arranged as either a row store or a column store. This choice is completely controlled by the configuration blurb we use. We populate the table with the data generated by the TPC `dbgen` utility. In the column store version, we use an appropriate dTable configuration for each column: a fixed-size dTable for columns storing floating point values, for instance, or an exception dTable above a small-integer

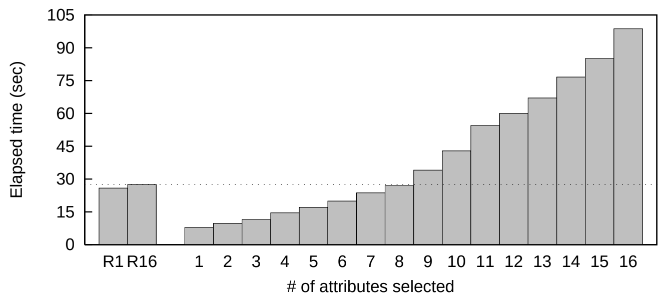


Figure 3.17: Time to select different numbers of columns from a row store (left bars; 1 and 16 columns) and a column store (right bars).

dTable above a fixed-size dTable for columns storing mostly small integers. (We can actually use an array dTable as the fixed-size dTable, since the keys are contiguous.) After populating the table, the column store is 807 MiB while the row store is 1008 MiB. (Using only linear dTables and without further configuration, the column store is 1334 MiB.) We then iterate through all the rows in the table, performing the Anvil equivalent of a simple SQL query of the form:

```
SELECT C1, C2, ... FROM lineitem WHERE pred(C1);
```

We vary the number of selected columns, using a predicate selecting 10% of the rows. We use the cTable iterator projection feature to efficiently select only the columns of interest in either a row or column store. The results, shown in Figure 3.17, are fairly similar to previous work [17], demonstrating that Anvil's modular design provides effective access to the same tradeoffs. For this test, it is important to drop both application-level and system-level caches between trials: otherwise, the (read-only) data is quickly cached in its entirety, and the CPU overhead of processing many more dTables in the column store dominates the "I/O" time after only two columns.

Sliding Window dTable Choosing a row or column store can tailor an Anvil configuration to specific query patterns, but dTable configurations can also be tuned to specific data. Many data sets, for instance, have the property that a few values are very common, and

	Digest (s)	Lookup (s)	Scan (s)	Size (MiB)
linear	3.7	53.6	7.9	209.8
sliding window	15.5	44.5	7.3	69.8

Figure 3.18: Sliding window dTable benchmark: various tasks and data file sizes with a sliding window dTable configuration or a plain linear dTable.

that many other values are uncommon. The sliding window dTable detects this and stores spatially local duplicate values only once, reducing the size of such data sets on disk. To evaluate its utility, we run a benchmark similar to the one used for the partition dTable. (In fact, that test was designed for the sliding window dTable, but reused for the partition dTable.) We use a managed dTable with a sliding window dTable, itself with fixed and indexed dTables for its key and value stores, respectively. We populate this configuration with 4 million values between 75 and 85 bytes, 75% of them being chosen from a set of 10 "popular" 80-byte values and the remaining 25% being generated randomly. We then digest the log, causing the sliding window dTable to compress that data and create the subordinate fixed and indexed dTables. Finally we do the same 1 million random key lookups and three forward iterator scans as before. In this version of the test, unlike the partition dTable version, we drop the file caches before each of these measurements: otherwise, we will not be able to see the difference in I/O time because the data was just written and is still in the caches. For comparison, we also run the same benchmark with a plain managed + linear dtable configuration.

Figure 3.18 shows the resulting times and data file sizes. Predictably, the sliding window dTable reduces the file sizes to about 33% of the uncompressed size, since nearly 75% of the values can be eliminated but there is some overhead to store pointers from the keys to the values. The time for the linear scans and random key lookups improves by 7.6% and 17.0%, respectively, showing that the decreased data size is beneficial when it must be read from disk. (The same tests, without dropping the caches first, instead show moderate performance degradation due to the additional CPU time involved.) Most striking, however, is that the time to digest – that is, the running time of the `create` procedure – is 4.2 times as large. Compressing data in this way involves non-trivial computation, and the price is paid at `create` time. Accordingly, this sort of expensive compression should only be used when the disk space or I/O time savings are required.

Originally, the sliding window dTable's `create` method was even more inefficient, due to one of Anvil's core design decisions: `create` methods accept iterators, and *pull* the data from them. Furthermore, many `create` methods (for instance those of the fixed and indexed dTables) scan the input iterator more than once. At first, the wrapper iterators that the sliding window dTable passed to the underlying `create` methods actually performed the compression. The data would therefore be compressed once for each scan, per subordinate dTable – five times in the above configuration (twice for the fixed dTable, and three times for the indexed dtable). While this design is very flexible for `create` methods, allowing them to avoid writing temporary files and easily precalculate summary information for use while writing their data files, it does not mix well with expensive conversions like compression. The sliding window dTable therefore opts to write temporary files anyway, allowing it to do the expensive compression step only once. The iterators it passes to the underlying `create` methods then merely read the temporary files, which is much less expensive than compressing the data. This pattern would likely be required for any other expensive conversions as well, for the same reason.

3.7.5 Digesting and Combining

Figure 3.19 shows the number of rows inserted per second (in thousands) while creating the row-based database used for the first two columns of Figure 3.17. Figure 3.20 shows the same operation, but with digests and combines run in the foreground, blocking other

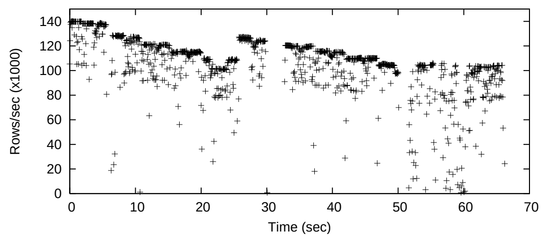


Figure 3.19: Rows inserted per second over time while creating the row-based TPC-H database, with digests and combines done in the background.

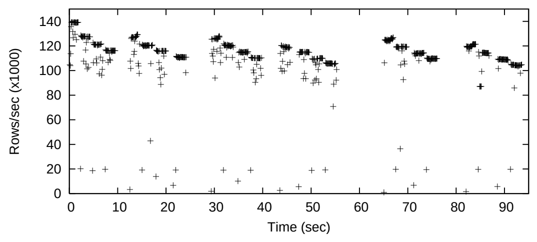


Figure 3.20: Rows inserted per second over time while creating the row-based TPC-H database. The discontinuities correspond to digests and combines done in the foreground.

progress. (Note that the *x* axes have different scales.) The periodic downward spikes in Figure 3.20 are due to digests, which take a small amount of time and therefore lower the instantaneous speed briefly. The longer periods of inactivity correspond to combine operations, which vary in length depending on how much data is being combined. In Figure 3.19, since these operations are done in the background, progress can still be made while they run.

Insertions become slower after each digest, since the row-based store must look up the previous row data in order to merge the new column data into it. (It does not know that the rows are new, although it finds out by doing the lookup.) After the combines, the speed increases once again, as there are fewer dTables to check for previous values. The effect is clearer when digests and combines are done in the foreground, as in Figure 3.20.

In this test, running digests and combines in the background takes about 27% less time than running them in the foreground. Most of our other benchmarks do not show such a significant improvement from background digests and combines; while there is still generally an improvement, it is much more modest (on the order of 5%). For this experiment, we configured the digests to occur with a very high frequency, to force them to occur enough times to have a performance effect on such a short benchmark. When a

Configuration	Time (s)	Overhead
No thread safety	77.6	N/A
C library thread safe, Anvil not	83.6	7.7%
Anvil thread-safe, C library not	88.8	14.4%
Both thread-safe	93.4	20.4%

Figure 3.21: Thread-safety overheads for TPC-H load: times to load the TPC-H data into Anvil, using foreground digesting and combining, with and without C library and internal Anvil thread-safety overheads.

background digest or combine is already in progress and the next digest is requested, the new digest request is ignored and the original background operation proceeds unchanged. As a result, fewer digests, and thus also fewer combines, occur overall. In more realistic configurations, background operations would overlap much less frequently with digest requests, and so the overall amount of work done would be closer to the same.

The entire database creation takes Anvil 67.9 seconds with background digesting and combining and 93.4 seconds without, both in delay mode. In comparison, loading the same data into unmodified SQLite in full and normal modes takes about 100 seconds, and about 64 seconds in async mode. Even though Anvil spends a large amount of time combining dTables, the managed dTable's digesting and combining schedule keeps this overhead in check. Further, the savings gained by contiguous disk access are larger than these overheads, and Anvil creates the database nearly as quickly as unmodified SQLite's async mode does.

Finally, as a measurement of the overhead imposed by thread safety – both that internal to Anvil and that automatically imposed by the C library after the first thread is created – we also run this experiment without even creating the background thread, and with Anvil's thread safety features disabled. The results of this experiment are shown in Figure 3.21. The overheads are significant: about 20% for both together. Although we have not done so, it should be possible to use a separate process rather than a thread for Anvil's background digesting and combining. If the application does not itself require thread-safe access, then

Transaction type	Time (s)
Basic	61.9
Abortable	85.5
Abortable (no optimization)	99.9

Figure 3.22: Comparison of Anvil's basic and abortable transactions for a simple test inserting 40 million keys. Without the optimization, abortable transactions are much slower.

one or both of these overheads could likely be eliminated in this way.

3.7.6 Abortable Transactions

Section 3.6 describes Anvil's abortable transaction mechanism, and notes that although it is elegantly modular, it does introduce additional cost compared to Anvil's basic transactions. In this section we evaluate the performance of this mechanism in two ways: first, by comparing the performance of basic transactions and abortable transactions in isolation, and second, by measuring the performance of basic transactions in the presence of abortable transactions.

For the first benchmark, we use a simple dTable configuration consisting of a managed dTable and linear dTable, and insert 40 million random keys and values. During the test we also run digest and combine operations, and periodically filter the system journal. We run three variants of the test: one where all insertions occur in basic transactions, and two where they are instead inserted using abortable transactions – one with the journal dTable optimization (§3.6) and one without. The results appear in Figure 3.22; abortable transactions are about 38% slower when using the optimization, and 61% without it.

The difference the optimization makes provides some insight into where the remaining cost of abortable transactions lies; it is primarily the cost to perform the rollover in memory. The data written to the disk is nearly identical, save one small entry to record the rollover event when committing an abortable transaction. A simple experiment confirmed this hypothesis: we disable the rollover itself, but still perform all the other work associated with abortable transactions, and find that the resulting system (while broken) performs as well as the basic version. (Technically, the rollover is not the *additional* work required for abortable transactions – the additional work is the initial insertions into the per-transaction journal dTables. The insertions done as part of the rollover would have been done anyway as part of a basic transaction.)

Based on the design of abortable transactions, and the results of the previous benchmark, we expect that the performance of basic transactions should be unaffected by the presence of abortable transactions. To test this, we run a series of very similar benchmarks, varying the proportion of basic and abortable transactions from 0 to 100% (in increments of 10%). We compute the average time to complete each type of transaction, and find that, as expected, they are invariant even as the proportions vary.

3.7.7 ACID Transactions

The ACID transaction dTable introduces still more cost on top of abortable transactions; to measure this overhead, we run a fourth variant of the previous benchmark, adding the ACID transaction dTable on top of abortable transactions. Since this benchmark uses only one transaction at a time, and never performs reads, it does not exercise the ACID transaction dTable's ability to abort transactions. (Separate correctness-checking unit tests do, however.) The benchmark does, however, measure the overhead involved in keeping the per-transaction and global state used to detect conflicting transactions. The results are shown in Figure 3.23.

While hardly negligible, the cost is within reason at about 26% overhead – less than that introduced by abortable transactions. Compared directly to using only basic transactions, the overhead for full ACID transactions is about 74%. There is likely room for

Transaction type	Time (s)
Abortable	85.5
ACID	107.4

Figure 3.23: Performance of Anvil's ACID transaction dTable using the same benchmark as Figure 3.22, compared to the simpler abortable transactions upon which it builds.

improvement here, potentially by reducing the time spent looking up per-transaction state with more refined data structures. Even as it stands, this does not seem like an unacceptable cost for this feature: on the DBT2 benchmark, for example, Anvil is 5.3 times faster than the original back end; even conservatively assuming that SQLite with Anvil ACID transactions would be uniformly 74% slower, it would still be over 3 times faster than the original back end. Plus, because it is a modular feature, it is an optional cost when full ACID semantics are not required.

3.7.8 Consistency and Durability Tests

To test the correctness of Anvil's consistency mechanisms, we set up a column store of 500 rows and 50 columns. We store an integer in each cell of the table and initialize all 25,000 cells to the value 4000. Thus, the table as a whole sums to 100 million. We then pick a cell at random and subtract 100 from it, and pick 100 other cells at random and add 1 to each. We repeat this operation 2000 times, and end the Anvil transaction. We then run up to 500 such transactions, which would take about 3 minutes if we allowed it to complete.

Instead, after initializing the table, we schedule a kernel module to load after a random delay of between 0 and 120 seconds. The module, when loaded, immediately reboots the machine without flushing any caches or completing any in-progress I/O requests. When the machine reboots, we allow ext3 to recover its journal, and then start up Anvil so that it can recover as well. We then scan the table, summing the cells to verify that they are consistent. The consistency check also produces a histogram of cell values so that we can subjectively verify that progress consistent with the amount of time the test ran before being interrupted was made. (The longer the test runs, the more distributed the histogram will tend to be, up to a point.)

During each transaction, the table is only consistent about 1% of the time: the rest of the time, the sum will fall short of the correct total. As long as the transactions are working correctly, these intermediate states should never occur after recovery. Further, the histograms should approximately reflect the amount of time each test ran. The result of over 1000 trials matches these expectations.

Finally, as evidence that the test itself can detect incorrectly implemented transactions, we note that it did in fact detect several small bugs in Anvil. One, for instance, occasionally allowed transaction data to "leak" out before its containing transaction committed. The test generally found these low-frequency bugs after only a few dozen trials, suggesting that it is quite sensitive to transaction failures. It is worth noting that these bugs were all specific to the ext3 version of the Anvil transaction library, and were failures to correctly specify the desired dependencies via the implicit ext3 mechanisms discussed earlier. The explicitly-specified dependencies in the Featherstitch version were not affected.

As a durability test, we run a simpler test that inserts a random number of keys into a managed dTable, each in its own durable transaction. We also run digest and combine operations occasionally during the procedure. After the last key is inserted, and its transaction reported as durable, we use the reboot module mentioned above to reboot the machine. Upon reboot, we verify that the contents of the dTable are correct. As this experiment is able to specifically schedule the reboot for what is presumably the worst possible time (immediately after a report of durability), we only run 10 trials by hand and find that durability is indeed provided. Running the same test without requesting transaction durability reliably results in a consistent but outdated dTable.

3.8 Summary

Anvil builds structured data stores by composing the desired functionality from sets of simple dTable modules. Simple configuration changes can substantially alter how Anvil stores data, and when unique storage strategies are needed, it is easy to write new dTables. The overhead incurred by Anvil's modularity, while not completely negligible, is small in comparison to the performance benefits it can offer, both due to its use of separate write-optimized and read-only dTables and to the ability to use specialized dTables for efficient data storage. Our prototype implementation of Anvil is faster than SQLite's original back end based on B-trees when running the TPC-C benchmark with DBT2, showing that its performance is reasonable for realistic workloads. Further, we can easily customize it as a column store for a benchmark loosely based on TPC-H, showing that optimizing it for specific data is both simple and effective. Even the implementation of abortable transactions was simplified by this modular design, as the pieces required to isolate transactions and commit them atomically were readily available in modular form.

Chapter 4

Conclusion

This work investigates two major types of data storage software systems, file systems and databases, and explores ways in which each can be made more modular while preserving the essential property of consistency. We have taken two different approaches to achieve this modularity: first, in Featherstitch, we introduce a new first-class object allowing modules to communicate write ordering requirements between each other while remaining only loosely coupled. Second, in Anvil, we isolate all writing in the system to a small handful of dedicated modules, allowing most modules to deal exclusively with read-only data and to themselves be divided into read-only and create-only parts. These approaches have both been successful at dividing into modules the data storage system to which we have applied them, and can even offer performance benefits as well – giving both users and system designers more flexibility and control over their data.

Featherstitch's explicit patch abstraction provides a new way for storage system implementations to formalize the "write-before" relationship among buffered changes to stable storage. This separates the specification and enforcement of the desired dependencies, and allows many modules and even user applications to cooperate loosely while providing strong consistency guarantees. The resulting modular design also simplifies the implementation of consistency mechanisms like journaling and soft updates, and allows user applications to specify custom dependencies, via the patchgroup module, in addition to those generated within the storage system. Using the patchgroup interface allows the buffer cache more freedom to reorder writes without violating the application's needs, while simultaneously freeing the application from having to micromanage writes to disk. We present results showing that the performance of our prototype is usually at least as fast as native Linux file systems that provide similar consistency guarantees, and that using the patchgroup interface can significantly reduce both the total time and the number of writes required for a realistic workload.

Anvil, on the other hand, takes virtually the opposite approach: rather than threading primitives that deal with consistency throughout the system, most of the system is made read-only to sidestep consistency problems altogether. Dedicated writable dTables handle all writes in the system, and special unifying dTables combine these with the read-only dTables to provide the illusion of unified writable stores. Anvil builds complex data stores by composing the desired functionality from sets of simple dTable modules, allowing small configuration changes to substantially alter how it stores data and making it easy to add unique storage strategies by writing new dTables. While not completely negligible, the overhead incurred by this modular design is small in comparison to the performance and flexibility benefits it can offer. Our prototype implementation of Anvil is faster than SQLite's original back end based on B-trees when running the TPC-C benchmark with DBT2, showing that its performance is reasonable for realistic workloads. Further, we can easily customize it as a column store for a benchmark loosely based on TPC-H, showing that optimizing it for specific data is both simple and effective. Even the implementation of ACID transactions was simplified by this modular design, as the pieces required to isolate transactions and commit them atomically were readily available in modular form.

In conclusion, these prototype systems demonstrate that modularity need not be eschewed in data storage systems like file systems and database back ends in order to provide consistency. Rather, they can be decomposed into modular components without substantial performance penalties, and without sacrificing the critical consistency properties of the system. Finally, these systems show that modular designs can dramatically increase the ability of these systems to be reconfigured and customized, providing both performance improvements and useful new features with minimal incremental effort.

Bibliography

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD '06*, pages 671–682, 2006.

[2] Eric Anderson, Martin Arlitt, Charles B. Morrey III, and Alistair Veitch. DataSeries: an efficient, flexible data format for structured serial data. *SIGOPS Operating Systems Review*, 43(1):70–75, 2009.

[3] Don Steve Batory, J. R. Barnett, Jorge F. Garza, Kenneth Paul Smith, K. Tsukuda, C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988. ISSN 0098-5589.

[4] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET Workshop*, pages 107–141, July 1970.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. ISSN 0001-0782.

[6] Peter Alexander Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[7] Nathan Christopher Burnett. *Information and Control in File System Buffer Management*. PhD thesis, University of Wisconsin—Madison, July 2006.

[8] CDB Constant DataBase. `http://cr.yp.to/cdb.html` (retrieved January 2010).

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach,

Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[10] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proc. 2004 USENIX Annual Technical Conference, FREENIX Track*, pages 19–28, Boston, Massachusetts, June 2004.

[11] DBT2. `http://sourceforge.net/projects/osdldbt/` (retrieved January 2010).

[12] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[13] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. *Graph Drawing*, pages 483–484, 2001.

[14] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proc. SOSP '07*, pages 307–320, 2007.

[15] Eran Gal and Sivan Toledo. A transactional Flash file system for microcontrollers. In *Proc. 2005 USENIX Annual Technical Conference*, pages 89–104, Anaheim, California, April 2005.

[16] Gregory R. Ganger, Marshall K. McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000.

[17] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proc. VLDB '06*, pages 487–498, 2006.

[18] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. SIGMOD '08*,

pages 981–992, 2008.

[19] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[20] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proc. USENIX Winter 1994 Technical Conference*, pages 235–246, San Francisco, California, January 1994.

[21] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint-Malô, France, October 1997.

[22] J Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997. `http://tinyurl.com/27ommd` (retrieved January 2010).

[23] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. USENIX Summer 1986 Technical Conference*, pages 238–247, Atlanta, Georgia, 1986.

[24] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Transactions on Database Systems*, 33(3):1–33, 2008. ISSN 0362-5915.

[25] Bruce Lindsay, John McPherson, and Hamid Pirahesh. A data management extension architecture. *SIGMOD Record*, 16(3):220–226, 1987. ISSN 0163-5808.

[26] Barbara Liskov and Rodrigo Rodrigues. Transactional file systems can be fast. In *Proc. 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[27] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In *Proc. SOSP '97*, pages 92–101, 1997.

[28] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on*

Computer Systems, 12(2):123–164, May 1994.

[29] Marshall K. McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the Fast Filesystem. In *Proc. 1999 USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, Monterey, California, June 1999.

[30] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[31] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proc. 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 115–128, San Francisco, California, March 2004.

[32] MySQL. `http://www.mysql.com/` (retrieved January 2010).

[33] MySQL Internals Custom Engine.
`http://forge.mysql.com/wiki/MySQL_Internals_Custom_Engine`
(retrieved January 2010).

[34] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles*, pages 191–205, Brighton, England, October 2005.

[35] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–14, Seattle, Washington, November 2006.

[36] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proc. 1999 USENIX Annual Technical Conference*, pages 43–43, Monterey, California, June 1999.

[37] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[38] Oracle. `http://www.oracle.com/` (retrieved January 2010).

[39] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proc. 1st USENIX Conference on File and Storage Technologies (FAST '02)*, pages 89–101, Monterey, California, January 2003.

[40] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[41] David S. H. Rosenthal. Evolving the Vnode interface. In *Proc. USENIX Summer 1990 Technical Conference*, pages 107–118, Anaheim, California, January 1990.

[42] Russell Sears and Eric Brewer. Stasis: flexible transactional storage. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 29–44, Seattle, Washington, November 2006.

[43] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. In *Proc. VLDB '08*, August 2008.

[44] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proc. 2000 USENIX Annual Technical Conference*, pages 71–84, June 2000.

[45] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, California, March 2003.

[46] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.

[47] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 239–252, San Francisco, California, December 2005.

[48] Glenn C. Skinner and Thomas K. Wong. "Stacking" Vnodes: A progress report.

In *Proc. USENIX Summer 1993 Technical Conference*, pages 161–174, Cincinnati, Ohio, June 1993.

[49] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, California, March 2003.

[50] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proc. 7th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 29–42, San Francisco, California, February 2009.

[51] SQLite. `http://www.sqlite.org/` (retrieved January 2010).

[52] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991. ISSN 0001-0782.

[53] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *Proc. VLDB '05*, pages 553–564, 2005.

[54] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (It's time for a complete rewrite). In *Proc. VLDB '07*, pages 1150–1160, 2007.

[55] Subversion. `http://subversion.tigris.org/` (retrieved January 2010).

[56] TPC-C. `http://www.tpc.org/tpcc/` (retrieved January 2010).

[57] TPC-H. `http://www.tpc.org/tpch/` (retrieved January 2010).

[58] Theodore Ts'o. Re: [evals] ext3 vs reiser with quotas, December 19 2004. `http://linuxmafia.com/faq/Filesystems/reiserfs.html` (retrieved January 2010).

[59] Theodore Ts'o. Delayed allocation and the zero-length file problem. Theodore Ts'o's blog. `http://tinyurl.com/dy7rgm` (retrieved January 2010).

[60] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *Proc. 4th Annual LinuxExpo*, Durham, North Carolina, 1998.

[61] UW IMAP toolkit. `http://www.washington.edu/imap/` (retrieved January 2010).

[62] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 17–30, San Francisco, California, December 2005.

[63] Mike Waychison. Re: fallocate support for bitmap-based files. linux-ext4 mailing list, June 29 2007. `http://www.mail-archive.com/linux-ext4@vger.kernel.org/msg02382.html` (retrieved January 2010).

[64] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3): 55–67, 2000.

[65] Charles P. Wright. *Extending ACID Semantics to the File System via ptrace*. PhD thesis, Stony Brook University, May 2006.

[66] Charles P. Wright, Michael C. Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proc. 2003 USENIX Annual Technical Conference*, pages 197–210, San Antonio, Texas, June 2003.

[67] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage*, March 2006.

[68] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathni. Using model checking to find serious file system errors. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, San Francisco, California, December 2004.

[69] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Seattle, Washington, November 2006.

[70] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proc. 2000 USENIX Annual Technical Conference*, pages 55–70, June 2000.

[71] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proc. 1999 USENIX Annual Technical Conference*, pages 57–70, Monterey, California, June 1999.

[72] ZFS Space Maps. `http://blogs.sun.com/bonwick/entry/space_maps` (retrieved January 2010).

[73] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.