Modular Data Storage with Anvil

Mike Mammarella
UCLA
mikem@cs.ucla.edu

Shant Hovsepian
UCLA
shant@cs.ucla.edu

Eddie Kohler
UCLA/Meraki
kohler@cs.ucla.edu

http://www.read.cs.ucla.edu/anvil/

ABSTRACT

Databases have achieved orders-of-magnitude performance improvements by changing the layout of stored data – for instance, by arranging data in columns or compressing it before storage. These improvements have been implemented in monolithic new engines, however, making it difficult to experiment with feature combinations or extensions. We present Anvil, a modular and extensible toolkit for building database back ends. Anvil's storage modules, called *dTables*, have much finer granularity than prior work. For example, some dTables specialize in writing data, while others provide optimized read-only formats. This specialization makes both kinds of dTable simple to write and understand. *Unifying* dTables implement more comprehensive functionality by layering over other dTables – for instance, building a read/write store from read-only tables and a writable journal, or building a general-purpose store from optimized special-purpose stores. The dTable design leads to a flexible system powerful enough to implement many database storage layouts. Our prototype implementation of Anvil performs up to 5.5 times faster than an existing B-tree-based database back end on conventional workloads, and can easily be customized for further gains on specific data and workloads.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; H.2.2 [**Database Management**]: Physical Design

General Terms: Design, Performance

Keywords: databases, software architecture, modular design

1 INTRODUCTION

Database management systems offer control over how data is physically stored, but in many implementations, ranging from embeddable systems like SQLite [23] to enterprise software like Oracle [19], 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, less-flexible interfaces like MySQL's custom storage engines [16]. Even recent specialized engines [7, 26] – which have shown significant benefits from data format changes, such as arranging data in columns instead of the traditional rows [11, 24] or compressing sparse or repetitive data [1, 31] – 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.
Copyright 2009 ACM 978-1-60558-752-3/09/10 ...$10.00.

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 split 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 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 some important features, including full concurrency and aborting transactions, 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.

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.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes Anvil's general design. Section 4 describes the Anvil transaction library, which provides the rest of the system with transactional primitives. Section 5 describes many of Anvil's individual dTables. Finally, we evaluate the system in Section 6, discuss future work in Section 7, and conclude in Section 8.

2 RELATED WORK

In the late 1980s, extensible database systems like Genesis [2] and Starburst [13] 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 [15] 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 [25] supported (and now PostgreSQL supports) user-defined index types, but these cannot control the physical layout of the data itself.

Monet [5] 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 [21] 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.

Anvil's split between read-only and write-mostly structures relates to read-optimized stores [11, 22] and log-structured file systems [20]. 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 [6, 18, 22]. 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 [10, 26].

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 [24]. 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 [11, 24],

Bigtable [7], 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 aborting transactions, overlapping transactions, and fine-grained locking to support high concurrency. 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.

```
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);      // create time
  };
  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 1: Simplified pseudocode for the dTable interface.

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.

3.1 dTables

dTables implement the key-value store interface summarized in Figure 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 [34]. The creation procedure 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

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

Figure 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.)

even. Specific examples of read-only dTables are presented in more detail in Section 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 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 5.5. Figure 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 which 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 5.2 and 5.4.

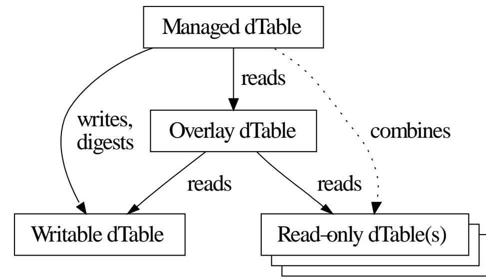

Figure 3: The relationships between a managed dTable and the dTables it uses.

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 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 [32] 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 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 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 [24], 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.

The managed dTable also maintains metadata describing which other dTables it is currently using and in what capacity. Metadata

```
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 4: A simplified, pseudocode version of the cTable interface.

updates are included in atomic transactions when necessary (using the transaction library described later), largely freeing other dTables from this concern.

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 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 *n* 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 6.3.

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]

[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.

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 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.

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 which 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.

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.

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 [14]. Anvil instead relies on file-system-specific mechanisms for enforcing orderings. By far the simpler of the mechanisms we've implemented is the explicit specification of ordering requirements using the Featherstitch storage system's *patchgroup* abstraction [9]. 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. However, Featherstitch is not widely deployed, and its implementation has several limitations we wished to avoid.

Anvil can therefore also use the accidental [30] 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 [27] 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 6.5 add confidence that our ext3-based consistency mechanisms are correct in the face of failures.

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 transactions only just before reporting success, as is done automatically in the xsyncfs file system [17].

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.

4.3 System Journal

Rather than using the transaction library directly, writable dTables use logging primitives provided by a shared logging facility, the

Class	dTable	Writable?	Description	Section
Storage	Linear	No	Stores arbitrary keys and values in sorted order	5.1
	Fixed-size	No	Stores arbitrary keys and fixed-size values in sorted order	5.4
	Array	No	Stores consecutive integer keys and fixed-size values in an array	5.4
	Unique-string	No	Compresses common strings in values	5.1
	Empty	No	Read-only empty dTable	5.1
	Memory	Yes	Non-persistent dTable	5.1
	Journal	Yes	Collects writes in the system journal	5.1
Performance	B-tree	No	Speeds up lookups with a B-tree index	5.2
	Bloom filter	No	Speeds up nonexistent key lookups with a Bloom filter	5.2
	Cache	Yes	Speeds up lookups with an LRU cache	5.2
Unifying	Overlay	No	Combines several read-only dTables into a single view	5.3
	Managed	Yes	Combines read-only and journal dTables into a read/write store	5.3
	Exception	No	Reroutes rejected values from a specialized store to a general one	5.5
Transforming	Small integer	No	Trims integer values to smaller byte counts	5.4
	Delta integer	No	Stores the difference between integer values	5.4
	State dictionary	No	Maps state abbreviations to small integers	5.4

Figure 5: Summary of dTables. Storage dTables write data on disk; all other classes layer over other dTables.

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.

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 [7]; 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 [33].

5 DTABLES

We now describe the currently implemented dTable types and their uses in more detail. The sixteen types are summarized in Figure 5. 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.

5.1 Storage dTables

The dTables described in this section store data directly on disk, rather than layering over other dTables.

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.

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.

B-tree dTable The *B-tree* dTable creates a B-tree [3] 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 [22] 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 [4] of the keys stored in an underlying read-only dTable. It responds to 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.

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.

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.

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 [32]. 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 which 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 We now have most of the basic mechanisms required to build a writable store from read-only and write-optimized pieces. The managed dTable handles the combination of these pieces, automatically coordinating the operation of subordinate dTables to hide the complexity of their interactions. For instance, all all other dTables can ignore transactions, leaving the managed dTable to take care of any transaction concerns using the transaction library.

The managed dTable stores its metadata as well as the files for its underlying dTables in a directory. It keeps a single journal dTable to which it sends all writes, and zero or more other dTables which, along with the journal dTable, are composed using an overlay dTable. Periodically, a special maintenance method digests the journal dTable to form a new read-only dTable, or combines several dTables to form a new single dTable. As in system journal cleaning, the actual data is written non-transactionally, but the transaction library is used to atomically "swap in" the newly digested or combined tables.

The current managed dTable schedules digest operations at fixed, configurable intervals. A digest will occur soon after the interval has elapsed. 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 is costly as well. 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 [12]. A more advanced version of the managed dTable would involve more carefully tuned parameters and might instead decide

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

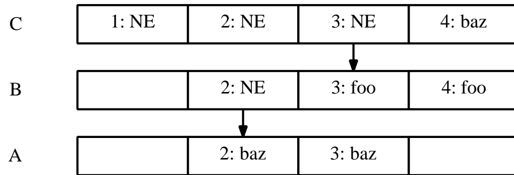

Figure 6: dTables C, B, and A in an overlay configuration; explicitly nonexistent values are shown as "NE." Digesting C should keep the nonexistent value for key 3, but not those for keys 1 and 2, since the combination of the other dTables already hold no values for those keys (B takes precedence over 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.

when to perform these tasks based on, for instance, the amount of data involved.

When digesting, an iterator for the journal dTable is passed to a suitable `create` method to create a new read-only dTable. When combining, an overlay dTable is created to merge the dTables to be combined, and an iterator for that is passed to the `create` method instead. To allow the omission of unnecessary nonexistent values, an additional dTable can be passed to `create` methods that contains all those keys that might need to be shadowed by the nonexistent values. The `create` method can look up a key with a nonexistent value in this "shadow dTable" to see if the nonexistent value is still required. The shadow dTable is simply an overlay dTable that merges all the dTables which are not being combined, but which the newly created dTable might shadow. Figure 6 shows an example set of dTables to help illustrate this algorithm.

Currently, client code is responsible for periodically calling a maintenance method, which in turn will trigger digesting and combining as the schedule requires. Most parts of Anvil are currently single-threaded; however, 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 6.4 evaluates the costs associated with digesting and combining, and quantifies the overhead imposed by using threads.

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. 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. Anvil currently provides a number of specialized dTables that efficiently store specific kinds of data.

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.

Small Integer dTable The *small integer* dTable is designed for values which 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 dictionary* 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.

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.

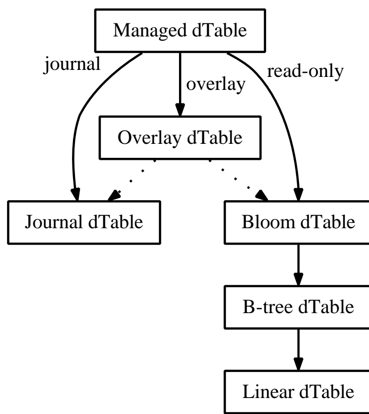

Figure 7: A simple dTable graph for the customer state example.

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 exception dTable is later created by digesting the temporary memory dTable.

5.6 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. First, suppose we want to store the 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 7.

While this arrangement will work, there are two properties of the data we haven't yet used to our advantage. First, the data is nearly all U.S. states, although some customers might live in other countries. We should therefore use the state dictionary dTable combined with an exception dTable for international customers. We could place these dTables under the B-tree dTable, but we instead insert the exception dTable directly under the Bloom filter dTable and use the B-tree dTable as its generic dTable. The reason is related to the final property of the data we want to use: the customer IDs are mostly sequential, so we can store the data in an array dTable much more efficiently. We therefore use the state dictionary dTable on top of the array dTable as the exception dTable's special-purpose dTable, and configure the array dTable with otherwise unused values to use to represent holes and nonexistent values.

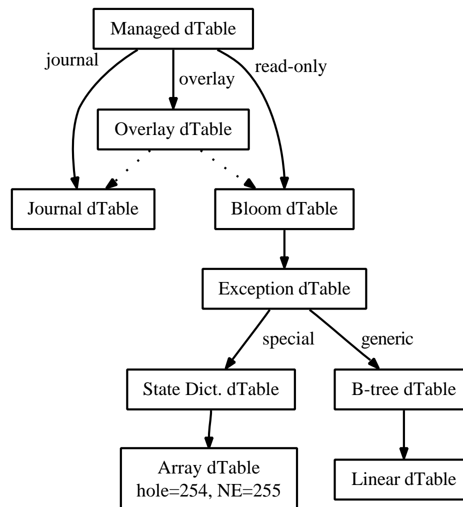

Figure 8: An example dTable graph for storing U.S. states efficiently, while still allowing other locations to be stored.

This configuration is shown in Figure 8, 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 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,

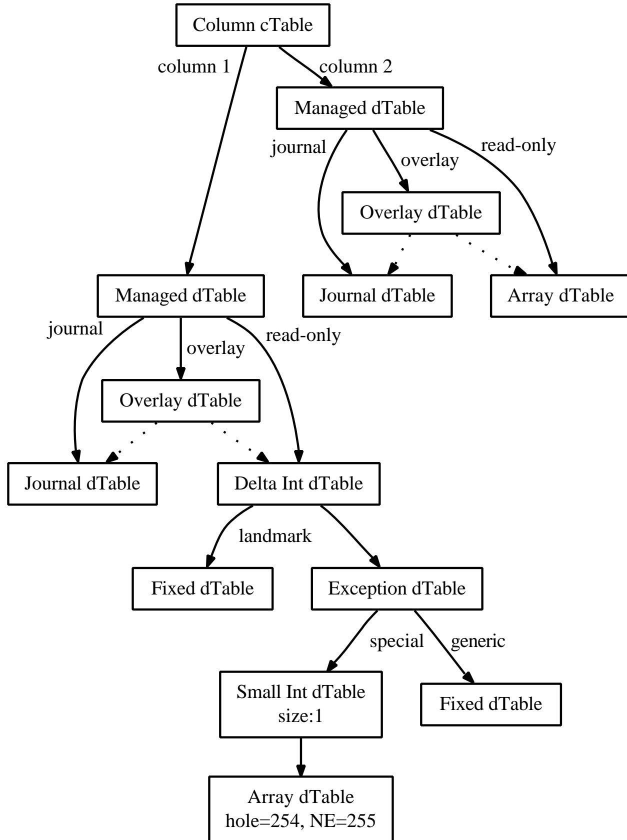

Figure 9: An example configuration for a cTable storing differentially timestamped log entries consisting of fixed-size customer IDs.

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 9.

6 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 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 10 GB 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 five runs.

	TPM	Disk util	Reqsz	W/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
Anvil, fsync	5066	31.4%	24.68	1077.5
Anvil, delay	8185	3.2%	439.60	9.7

Figure 10: Results from running the DBT2 test suite. *TPM* represents "new order Transactions Per Minute"; larger numbers are better. *Disk util* is disk utilization, *Reqsz* the average size in KiB of the issued requests, and *W/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.

6.1 Conventional Workload

For our conventional workload, we use the DBT2 [8] test suite, which is a "fair usage implementation"[3] of the TPC-C [28] 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 benchmark as Anvil does not yet support rolling back transactions. 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 run SQLite with Anvil in two different modes: *fsync*, which matches the durability guarantee of the original *full* mode by calling `fsync` at the end of every transaction, and *delay*, which allows larger group commits as described in Section 4.2. Both of these modes, as well as the first two unmodified SQLite modes, provide consistency in the event of a crash; SQLite's async mode does 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 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. The performance advantage of read- and write-optimized structures far outweighs any cost of separating these functions into separate modules.

6.2 Microbenchmarks

We further evaluate the performance consequences of Anvil's modularity by stress-testing Anvil's most characteristic modules,

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

	Time (s)		
	Digest	Lookup	Size (MiB)
linear	2.03	63.37	49.6
btree	2.45	23.58	80.2
array	1.59	8.81	22.9
excep+array	1.71	9.15	23.0
excep+fixed	2.09	56.46	34.4
excep+btree+fixed	2.50	23.87	65.0

Figure 11: 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 4% of the specialized version while supporting any value type.

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 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 11.

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 only approximately 3.9% 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.)

	Lookup	Scan
direct	140.2 s	13.95 s
overlay	147.9 s	15.34 s
overhead	5.49%	9.96%

Figure 12: 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.

	Time (s)		
	Even keys	Odd keys	Mixed keys
direct	30.36	24.95	27.70
bloom	32.08	1.05	16.63

Figure 13: 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.

The results are shown in Figure 12. 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 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.

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 13. The Bloom filter dTable adds about 5.6% 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 4% 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.

6.3 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 [29], 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 [11, 22]. We adapt the method of Harizopoulos et al. [11], as it does not require building a query language or relational algebra.

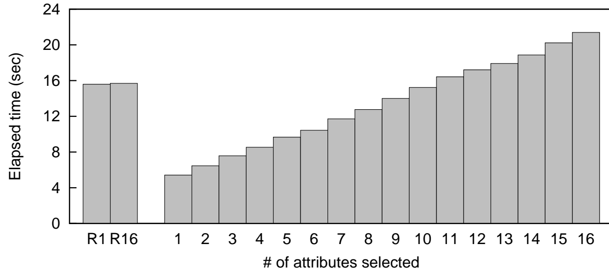

Figure 14: Time to select different numbers of columns from a row store (left bars; 1 and 16 columns) and a column store (right bars).

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 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 910 MiB while the row store is 1024 MiB (without customizations, 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, ... 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 14, are very similar to previous work [11], demonstrating that Anvil's modular design provides effective access to the same tradeoffs.

6.4 Digesting and Combining

Figure 15 shows the number of rows inserted per second (in thousands) while creating the row-based database used for the first two columns of Figure 14. Figure 16 shows the same operation, but with digests and combines run in the foreground, blocking other progress. (Note that the *x* axes have different scales.) The periodic downward spikes in Figure 16 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 15, 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 16.

In this test, running digests and combines in the background takes about 40% 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 or 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 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.

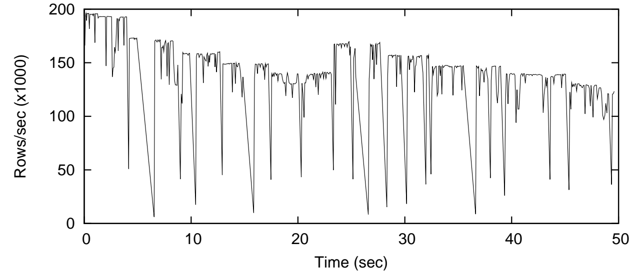

Figure 15: Rows inserted per second over time while creating the row-based TPC-H database, with digests and combines done in the background.

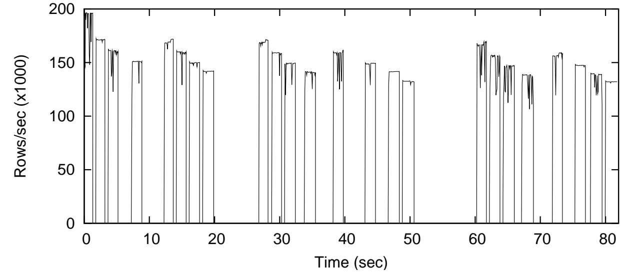

Figure 16: Rows inserted per second over time while creating the row-based TPC-H database. The discontinuities correspond to digests and combines done in the foreground.

The entire database creation takes Anvil about 50 seconds with background digesting and combining and 82 seconds without, both in delay mode. In comparison, loading the same data into unmodified SQLite in async mode takes about 64 seconds, and about 100 seconds in full and normal modes. 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 slightly faster overall for this test.

Finally, as a measurement of the overhead automatically imposed by the C library to make itself thread-safe after the first thread is created, we also run this experiment without even creating the background thread. In this configuration, the load takes about 77 seconds, indicating that the C library thread-safety overhead is about 6% for this workload. Using a separate process rather than a thread should eliminate this overhead.

6.5 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.

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.

7 Future Work

There are several important features that we have not yet implemented in Anvil, but we do have plans for how they could be added. In this section, we briefly outline how two of these features, abortable transactions and independent concurrent access, could be implemented within the Anvil design.

Abortable transactions Anvil's modular dTable design may facilitate, rather than hinder, abortable transactions. Each abortable transaction could create its own journal dTable in which to store its changes, and use local overlay dTables to layer them over the "official" stores. This should be a small overhead, 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 1.24 million such pairs per second on our benchmark machine.) To abort the transaction, these temporary journals would be discarded and removed from the corresponding managed dTables. To commit, the journal data would instead be folded into the official journal dTables by writing small records to that effect to the system journal and merging the in-memory balanced trees. The transaction's data would later to be digested as usual.

Independent concurrent access The design for abortable transactions already contains part of the mechanism required for independent concurrent access. Different transactions would need independent views of the dTables they are using, with each transaction seeing only its changes. By creating a separate journal dTable and overlay dTable for each transaction, and having more than one such transaction at a time, the different transactions would automatically be isolated from each other's changes. Independent transactions could be assigned IDs on request or automatically. Committing one of the transactions could either roll its changes into the official journal dTable immediately, making those changes visible to other transactions, or append its journal dTable to the official list and merge the changes once no other transactions require a view of the data before the commit. The major challenges in implementing this proposal would seem to be shared with any system with concurrent transactions, namely detecting conflicts and adding locks. Unfortunately, some concurrency disciplines seem perhaps difficult to add as separate modules; for example, every storage dTable might require changes to support fine-grained record locking.

8 Conclusion

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. While Anvil still lacks some important features, they do not appear to be fundamentally precluded by our design.

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.

Acknowledgments

We would like to thank the members of our lab, TERTL, for sitting through several meetings at which ideas much less interesting than those in this paper were discussed at length. In particular, we would like to thank Petros Efstathopoulos, whose comments on early versions of this paper inspired several useful major changes, and Steve VanDeBogart, who modified DBT2 to work with SQLite (allowing us to run TPC-C). We would also like to thank the anonymous reviewers and our shepherd, David Andersen, for the time they dedicated to providing valuable feedback on drafts of this paper. Our work was supported by the National Science Foundation under Grant Nos. 0546892 and 0427202; by a Microsoft Research New Faculty Fellowship; by a Sloan Research Fellowship; and by an equipment grant from Intel. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Finally, we would like to thank our lab turtles, Vi and Emacs, for being turtles in a lab whose acronym is homophonic with the name of their species, and for never having complained about their names.

REFERENCES

[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] 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.

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

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

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

[6] CDB Constant DataBase. `http://cr.yp.to/cdb.html`.

[7] 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. OSDI '06*, pages 205–218, November 2006.

[8] DBT2. `http://sf.net/projects/osdldbt/`.

[9] 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.

[10] 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.

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

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

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

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

[15] MySQL. `http://www.mysql.com/`.

[16] MySQL Internals Custom Engine. `http://forge.mysql.com/wiki/MySQL_Internals_Custom_Engine`.

[17] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. OSDI '06*, pages 1–14, November 2006.

[18] 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.

[19] Oracle. `http://www.oracle.com/`.

[20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):1–15, 1992.

[21] Russell Sears and Eric Brewer. Stasis: flexible transactional storage. In *Proc. OSDI '06*, pages 29–44, November 2006.

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

[23] SQLite. `http://www.sqlite.org/`.

[24] 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.

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

[26] 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.

[27] Subversion. `http://subversion.tigris.org/`.

[28] TPC-C. `http://www.tpc.org/tpcc/`.

[29] TPC-H. `http://www.tpc.org/tpch/`.

[30] Theodore Ts'o. Delayed allocation and the zero-length file problem. Theodore Ts'o's blog. `http://tinyurl.com/dy7rgm` (retrieved March 2009).

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

[32] 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*, 2(1):74–105, February 2006.

[33] ZFS Space Maps. `http://blogs.sun.com/bonwick/entry/space_maps`.

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