Story Book: An Efficient Extensible Provenance Framework

Appears in the Proceedings of the 1st USENIX Workshop on Theory and Practice of Provenance

R. Spillane, R. Sears*, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok
*Stony Brook University and *University of California, Berkeley*

Abstract

Most application provenance systems are hard coded for a particular type of system or data, while current provenance file systems maintain in-memory provenance graphs and reside in kernel space, leading to complex and constrained implementations. Story Book resides in user space, and treats provenance events as a generic event log, leading to a simple, flexible and easily optimized system.

We demonstrate the flexibility of our design by adding provenance to a number of different systems, including a file system, database and a number of file types, and by implementing two separate storage backends. Although Story Book is nearly 2.5 times slower than ext3 under worst case workloads, this is mostly due to FUSE message passing overhead. Our experiments show that coupling our simple design with existing storage optimizations provides higher throughput than existing systems.

1 Introduction

Existing provenance systems are designed to deal with specific applications and system architectures, and are difficult to adapt to other systems and types of data. Story Book decouples the application-specific aspects of provenance tracking from dependency tracking, queries and other mechanisms common across provenance systems.

Story Book runs in user space, simplifying its implementation, and allowing it to make use of existing, off-the-shelf components. Implementing provenance (or any other) file system in user space incurs significant overhead. However, our user space design significantly reduces communication costs for Story Book's application-specific provenance extensions, which may use existing IPC mechanisms, or even run inside the process generating provenance data.

Story Book supports application-specific extensions, allowing it to bring provenance to new classes of systems. It currently supports file system and database provenance and can be extended to other types of systems, such as Web or email servers. Story Book's file system provenance system also supports extensions that record additional information based on file type. We have implemented two file-type modules: One records the changes made by applications to `.txt` files, and the other records modifications to `.docx` metadata, such as the author name.

Our experiments show that Story Book's storage and query performance are adequate for file system provenance workloads where each process performs a handful of provenance-related actions. However, systems that service many small requests across a wide range of users, such as databases and Web servers, generate provenance information at much higher rates and with finer granularity. As such systems handle most of today's multi-user workloads, we believe they also provide higher-value provenance information than interactive desktop systems. This space is where Story Book's extensible user-space architecture and high write throughput are most valuable.

The rest of this paper is organized as follows: Section 2 describes existing approaches to provenance, Section 3 describes the design and implementation of Story Book, and Section 4 describes our experiments and performance comparisons with PASSv2. We conclude in Section 5.

2 Background

A survey of provenance systems [14] provides a taxonomy of existing provenance databases. A survey of the taxonomy shows that Story Book is flexible enough to cover much of the design space targeted by existing systems. This flexibility comes from Story Book's layered, modular approach to provenance tracking (Figure 1).

A *provenance source* intercepts user interaction events with application data and sends these events to *application specific extensions* which interpret them and generate provenance inserts into one of Story Book's *storage backends*. Queries are handled by Story Book's *Story Book API*. Story Book relies on external libraries to implement each of its modules. FUSE [8] and MySQL [18] intercept file system and database events. Extensions to Story Book's FUSE file system, such as the `.txt` and `.docx` modules, annotate events with application-specific provenance. These provenance records are then inserted into either Stasis [12] or Berkeley DB [15]. Stasis stores provenance

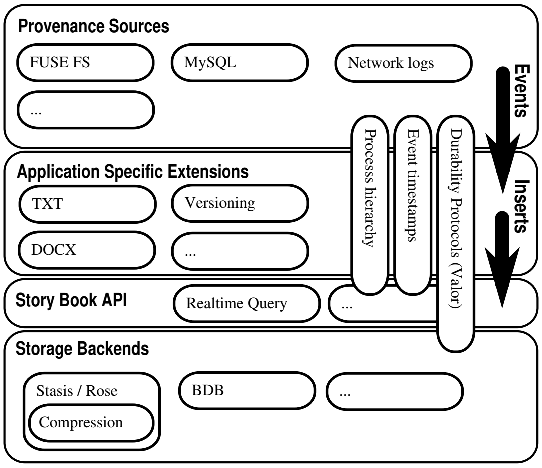

Figure 1: Story Book's modular approach to provenance tracking.

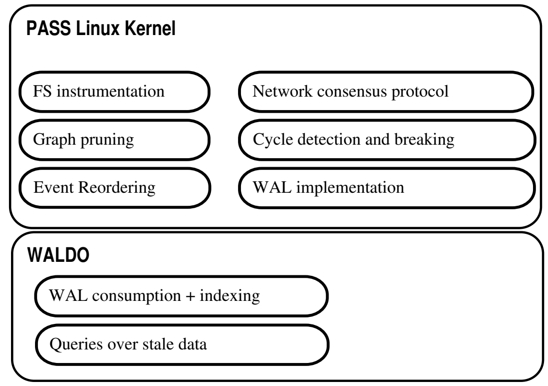

Figure 2: PASSv2's monolithic approach to provenance tracking. Although Story Book contains more components than PASSv2, Story Book addresses a superset of PASSv2's applications, and leverages existing systems to avoid complex, provenance-specific code.

data using database-style no-Force/Steal recovery for its hashtables, and compressed log structured merge (LSM) trees [11] for its Rose [13] indexes. Story Book utilizes Valor [16] to maintain write-ordering between its logs and the kernel page cache. This reduces the number of disk flushes, greatly improving performance.

Although Story Book utilizes a modular design that separates different aspects of its operation into external modules (i.e., Rose, Valor, and FUSE), it does hardcode some aspects of its implementation. Simhan's survey discusses provenance systems that store metadata separately from application data, and provenance systems that do not. One design decision hardcoded into Story Book is that metadata is always stored separately from application data for performance reasons. Story Book stores blob data in a physically separate location on disk in order to preserve locality of provenance graph nodes and efficiently support queries.

Story Book avoids hardcoding most of the other design decisions mentioned by the survey. Because Story Book does not directly support versioning, application-specific provenance systems are able to decide whether to store logical undo/redo records of application operations to save space and improve performance, or to save raw value records to reduce implementation time. Story Book's `.txt` module stores patches between text files rather than whole blocks or pages that were changed.

Story Book allows applications to determine the granularity of their provenance records, and whether to focus on information about data, processes or both. Similarly, Story Book's implementation is independent of the format of extended records, allowing applications to use metadata standards such as RDF [19] or Dublin Core [2] as they see fit.

The primary limitation of Story Book compared to systems that make native use of semantic metadata is that Story Book's built-in provenance queries would need to be extended to understand annotations such as "version-of" or "derived-from," and act accordingly. Applications could implement custom queries that make use of these attributes, but such queries would incur significant I/O cost. This is because our schema stores (potentially large) application-specific annotations in a separate physical location on disk. Of course, user-defined annotations could be stored in the provenance graph, but this would increase complexity, both in the schema, and in provenance query implementations.

Databases such as Trio [20] reason about the quality or reliability of input data and processes that modify it over time. Such systems typically cope with provenance information, as unreliable inputs reflect uncertainty in the outputs of queries. Story Book targets provenance over exact data. This is especially appropriate for file system provenance and regulatory compliance applications; in such circumstances, it is more natural to ask which version of an input was used rather than to calculate the probability that a particular input is correct.

2.1 Comparison to PASS

Existing provenance systems couple general provenance concepts to system architectures, leading to complex in-kernel mechanisms and complicating integration with application data representations. Although PASSv2 [9] suffers from these issues, it is also the closest system to Story Book (Figure 2).

PASSv2 distinguishes between data *attributes*, such as name or creation time, and *relationships* indicating things like data flow and versioning. Story Book performs a similar layering, except that it treats versioning information (if any) as an attribute. Furthermore, whereas PASSv2 places such mechanisms in kernel space, Story Book places them in user space.

On the one hand, this means that user-level provenance inspectors could tamper with provenance information; on the other it has significant performance and usability advantages. In regulatory environments (which cope with untrusted end-users), Story Book should run in a trusted file or other application server. The security implications of this approach are minimal: in both cases a malicious user requires root access to the provenance server in order to tamper with provenance information.

PASSv2 is a *system-call-level* provenance system, and requires that all provenance data pass through relevant system calls. Application level provenance data is tracked by applications, then used to annotate corresponding system calls that are intercepted by PASSv2. In contrast, Story Book is capable of system call interception, but does not require it. This avoids the need for applications to maintain in-memory graphs of provenance information. Furthermore, it is unclear how system-call-level instrumentation interacts with mechanisms such as database buffer managers which destroy the relationship between application-level operations and system calls. Story Book's ability to track MySQL provenance shows that such systems are easily handled with our approach.

PASSv2 supersedes PASS, which directly wrote provenance information to a database. The authors of the PASS systems concluded that direct database access was neither "efficient nor scalable," and moved database operations into an asynchronous background thread.

Our experiments show that, when properly tuned, Story Book's *Fable* storage system and PASSv2's *Waldo* storage system both perform well enough to allow direct database access, and that the bulk of provenance overhead is introduced by system instrumentation, and insertion-time manipulation of provenance graphs. Story Book suffers from the first bottleneck while PASSv2 suffers from the latter.

This finding has a number of implications. First, Story Book performs database operations synchronously, so its provenance queries always run against up-to-date information. Second, improving Story Book performance involves improvements to general-purpose code, while PASSv2's bottleneck is in provenance-specific mechanisms.

From a design perspective then, Story Book is at a significant advantage; improving our performance is a matter of utilizing a different instrumentation technology. In contrast, PASSv2's bottlenecks stem from a special-purpose component that is fundamental to its design.

PASSv2 uses a recovery protocol called *write ahead provenance*, which is similar to Story Book's file-system recovery approach. However, Story Book is slightly more general, as its recovery mechanisms are capable of entering into system-specific commit protocols. This is important when tracking database provenance, as it allows Story Book to reuse existing durability mechanisms, such as Valor, and inexpensive database replication techniques. For our experiments, we extended write ahead provenance with some optimizations employed by Fable so that PASSv2's write throughput would be within an order of magnitude of ours.

PASSv2 supports a number of workloads currently unaddressed by Story Book, such as network operation and unified naming across provenance implementations. The mechanisms used to implement these primitives in PASSv2 follow from the decision to allow multiple provenance sources to build provenance graphs before applying them to the database. Even during local operation, PASSv2 employs special techniques to avoid cycles in provenance graphs due to reordering of operations and record suppression. In the distributed case, clock skew and network partitions further complicate matters.

In contrast, Story Book appends each provenance operation to a unified log, guaranteeing a consistent, cycle-free dependency graph (see Section 3.8). We prefer this approach to custom provenance consensus algorithms, as the problem of imposing a partial order over events in a distributed system is well-understood; building a distributed version of Story Book is simply a matter of building a provenance source that makes use of well-known distributed systems techniques such as logical clocks [6] or Paxos [7].

3 Design and Implementation

Story Book has two primary scalability goals: (1) decrease the overhead of logging provenance information, and (2) decrease the implementation effort required to track the provenance of applications which use the file system or other instrumentable storage to organize data.

We discuss Story Book's application development model in Section 3.1, and Story Book's architecture and provenance file system write path in Section 3.2. We explain Story Book's indexing optimizations in Section 3.3 and recovery in Section 3.4. We discuss reuse of third party code in Section 3.5 and the types of provenance that are recorded on-disk in Section 3.6. We explain how Story Book accomplishes a provenance query in Section 3.7 and describe Story Book's ability to compress and suppress redundant provenance information in Section 3.8.

3.1 Application Development Model

Story Book provides a number of approaches to application-specific provenance. The most powerful, efficient and precise approach directly instruments applications, allowing the granularity of operations and actors recorded in the provenance records to match the application. This approach also allows for direct communication between the application and Story Book, minimizing communication overheads.

However, some applications cannot be easily instrumented, and file formats are often modified by many different applications. In such cases, it makes sense to track application-specific provenance information at the file system layer. Story Book addresses this problem by allowing developers to implement *provenance inspectors*, which intercept requests to modify particular sets of files. When relevant files are accessed, the appropriate provenance inspector is alerted and allowed to log additional provenance information. Provenance inspectors work best when the intercepted operations are indicative of the application's intent (e.g., an update to a document represents an edit to the document). Servers that service many unrelated requests or use the file system primarily as a block device (e.g., Web servers, databases) are still able to record their provenance using Story Book's file-system provenance API, though they would be better-served by Story Book's application-level provenance.

For our evaluation of Story Book's file system-level provenance, we implemented a `.txt` and a `.docx` provenance inspector. The `.txt` inspector records patches that document changes made to a text file between the open and close. The `.docx` inspector records changes to document metadata (e.g., author names, title) between the time it was opened and closed. We describe provenance inspectors in more detail in Section 3.6.3.

3.2 Architecture

Story Book stores three kinds of records: (1) *basic records* which are a record of an open, close, read or write, (2) *process records* which are a record of a caller-callee relationship and (3) *extended records* which are a record of application-specific provenance information. These records are generated by the provenance sources and application specific extensions layers. The storage backend in Story Book which is optimized for write throughput by using Rose and Stasis is called *Fable*. Basic, process, and extended records are inserted into Fable, or into the Berkeley DB backend if it is being used.

Rather than synchronously commit new basic, process and extended records to disk before allowing updates to propagate to the file system, we use Stasis' non-durable commit mechanism and Valor's write ordering to ensure that write-ahead records reach disk before file system operations. This approach is a simplified version of Speculator's [10] *external synchrony* support, in that it performs disk synchronization before page write-back, but unlike Speculator, it will not sync the disk before any event which presents information to the user (e.g., via `tty1`).

This allows Fable to use a single I/O operation to commit batches of provenance operations just before the file system flushes its dirty pages. Although writes to Stasis' page file never block log writes or file system writeback, hash index operations may block if the Stasis pages they manipulate are not in memory. Unless the system is under heavy memory pressure, the hash pages will be resident. Therefore, most application requests that block on Fable I/O also block on file system I/O.

Figure 3 describes what happens when an application P writes to a document. The kernel receives the request from P and forwards the request to the FUSE file system (1). FUSE forwards the request to Story Book's FUSE daemon (2) which determines the type of file being accessed. Once the file type is determined, the request is forwarded to the appropriate provenance inspector (3). The provenance inspector generates basic and extended log records (4) and schedules them for asynchronous insertion into a write-ahead log (5). Fable synchronously updates (cached) hash table pages and Rose's in-memory component (6) and then allows the provenance inspector to return to the caller P (7). After some time, the file system flushes dirty pages (8), and Valor ensures that any scheduled entries in Fable's log are written (9) before the file system pages (10). Stasis' dirty pages are occasionally written to disk to enable log truncation or to respond to memory pressure (11).

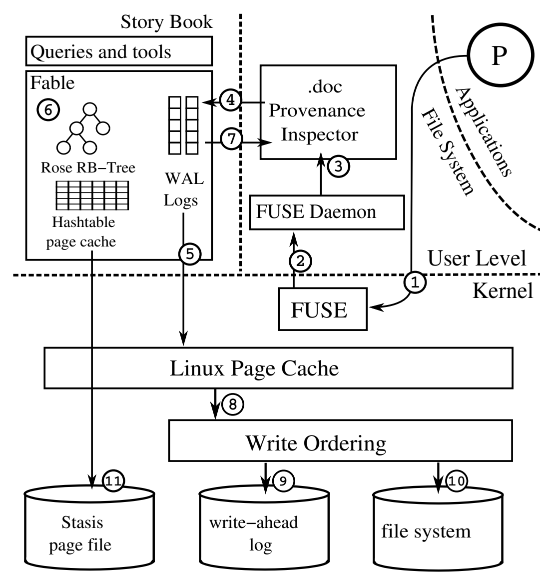

Figure 3: Write path of Story Book's provenance file system

3.3 Log Structured Merge Trees

Rose indexes are compressed LSM-trees that have been optimized for high-throughput and asynchronous, durable commit. Their contents are stored on disk in compressed, bulk loaded B-trees. Insertions are serviced by inserting data into an in-memory red-black tree. Once this tree is big enough, it is merged with the smallest on-disk tree and emptied. Since the on-disk tree's leaf nodes are laid out sequentially on disk, this merge does not cause a significant number of disk seeks. Similarly, the merge produces data in sorted order, allowing it to contiguously lay out the new version of the tree on disk.

The small on-disk tree component will eventually become large, causing merges with in-memory trees to become prohibitively expensive. Before this happens, we merge the smaller on-disk tree with a larger on-disk tree, emptying the smaller tree. By using two on-disk trees and scheduling merges correctly, the amortized cost of insertion is reduced to $O(log\ n * \sqrt{n})$, with a constant factor proportional to the cost of compressed sequential I/O. B-trees' $O(log\ n)$ insertion cost is proportional to the cost of random I/O. The ratio of random I/O cost to sequential I/O cost is increasing exponentially over time, both for disks and for in-memory operations.

Analysis of asymptotic LSM-tree performance shows that, on current hardware, worst-case LSM-tree throughput will dominate worst-case B-tree write throughput across all reasonable index sizes, but that in the best case for a B-tree, insertions arrive in sorted order, allowing it to write back dirty pages serially to disk with perfect locality. Provenance queries require fast lookup of basic records based on inode, not arrival time, forcing Story Book's indexes to re-sort basic records before placing them on disk. If insertions do show good locality (due to skew based on inode, for example), then Rose's tree merges will needlessly touch large amounts of clean data. Partitioning the index across multiple Rose indexes would lessen the impact of this problem [4].

Sorted data compresses well, and Story Book's tables have been chosen to be easily compressible. This allows Rose to use simple compression techniques, such as run length encoding, to conserve I/O bandwidth. Superscalar implementations of such compression algorithms compress and decompress with GiB/s throughput on modern processor cores. Also, once the data has been compressed for writeback, it is kept in compressed form in memory, conserving RAM.

3.4 Recovery

As we will see in Section 3.6, the basic records stored in Fable contain pointers into extended record logs. At runtime, Fable ensures that extended records reach disk before the basic records that reference them. This ensures that all pointers from basic records encountered during recovery point to complete extended records, so recovery does not need to take any special action to ensure that the logs are consistent.

If the system crashes while file system (or other application) writes are in flight, Story Book must ensure that a record of the in-flight operations is available in the provenance records so that future queries are aware of any potential corruption due to crash.

Since Valor guarantees that Stasis' write ahead log entries reach disk before corresponding file system operations, Fable simply relies upon Stasis recovery to recover from crash. Provenance over transactional systems (such as MySQL) may make use of any commit and replication mechanisms supported by the application.

3.5 Reusing User-Level Libraries

Provenance inspectors are implemented as user-level plugins to Story Book's FUSE daemon. Therefore, they can use libraries linked by the application to safely and efficiently extract application-specific provenance. For example, our `.docx` inspector is linked against the XML parser ExPat [3]. Story Book's FUSE-based design facilitates transparent provenance inspectors that require no application modification and do not force developers to port user-level functionality into the kernel [17].

3.6 Provenance Schema

Story Book models the system's provenance using basic records, process records and extended records. To support efficient queries, Fable maintains several different Rose trees and persistent hash indexes. In this section we discuss the format of these record types and indexes.

3.6.1 Basic Records

Story Book uses basic records to record general provenance. To determine the provenance or history of an arbitrary file, Story Book must maintain a record of which processes read from and wrote to which objects. Story Book clients achieve this by recording a basic record for every create, open, close, read and write regardless of type. The format of a basic record is as follows:

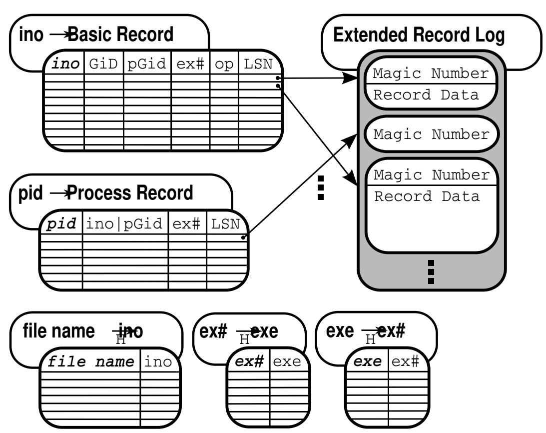

Figure 4: Story Book Database Layout. Table keys are bold. 'H' indicates the table is a hash table.

GLOBAL ID The ID of the process performing the operation. This ID is globally unique across reboots.

PARENT GLOBAL ID The global ID of the parent of the process performing the operation.

INODE The inode number of a file or object id.

EXECUTABLE ID The global ID corresponding to the absolute path of the executable or other name associated with this process.

OPERATION Indicates if the process performed a read, write, open or close.

LSN The log sequence number of the record, used for event ordering and lookup of application-specific extended records

Figure 4 illustrates how Fable stores basic records. Executable paths are stored using two hash tables that map between executable ID's and executable names ('`exe`' to '`ex#`') and ('`ex#`' to '`exe`'). The first is used during insertion; the second to retrieve executable names during queries. File names are handled in a similar fashion. Fable stores the remaining basic record attributes in a Rose table, sorted by inode / object id.

3.6.2 Process Records

Story Book uses process records to record parent-child relationships. When processes perform file operations Story Book records their executable name and process ID within a basic record. However, Story Book cannot rely on every process to perform a file operation (e.g., the `cat` program in Figure 5), and must also record *process records* alongside basic records and application-specific provenance. Process records capture the existence of processes that might never access a file. The format of a process record is as follows:

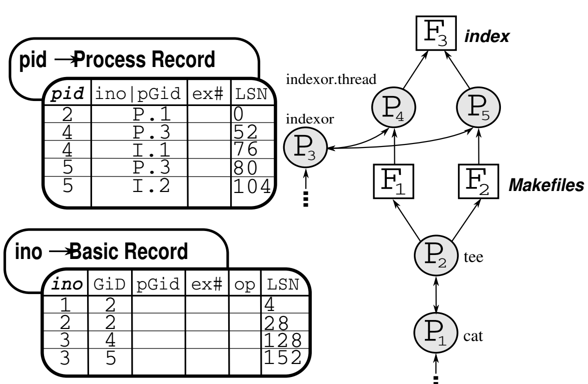

Figure 5: A provenance query. On the left is the data used to construct the provenance graph on the right. The query begins with a hash lookup on the file name's inode whose provenance we determine (< "`index`", 3 >).

GLOBAL ID The ID of the process performing the file access. This ID is globally unique across reboots.

PARENT GLOBAL ID OR FILE INODE Contains either the global ID of the parent of this process or the inode of a file this process modified or read.

EXECUTABLE ID Global ID corresponding to the absolute path of the executable.

LSN The log sequence number of the record, used to establish the ordering of child process creation and file operations during provenance queries.

Before Story Book emits a basic record for a process it first acquires all the global IDs of that process's parents by calling a specially added system call. It records a process record for every (`global ID`, `parent global ID`) pair it has not already recorded as a process record at some earlier time.

The exact mechanisms used to track processes vary with the application being tracked, but file system provenance is an informative example. By generating process records during file access rather than at `fork` and `exit`, file system provenance is able to avoid logging process records for processes that do not modify files. This allows it to avoid maintenance of a provenance graph in kernel RAM during operation, but is not sufficient to reflect process reparenting that occurs before a file access or a change in the executable name that occurs after the last file access.

3.6.3 Extended Records

Story Book manages application-specific provenance by having each application's provenance inspector attach additional application-specific information to a basic record. This additional information is called an *extended record*. Extended records are not stored in Rose, but rather are appended in chronological order to a log. This is because their format and length may vary and extended records are not necessary to reconstruct the provenance of a file in the majority of cases. However, the extended record log cannot be automatically truncated, since it contains the only copy of the provenance information stored in the extended records. An extended record can have multiple application-specific entries or none at all. The format of an extended record is as follows:

MAGIC NUMBER Indicates the type of the record data that is to follow. For basic or process records with no application-specific data to store this value is 0.

RECORD DATA Contains any application-specific data.

Each basic and process record must refer to an extended record because the offset of the extended record is used as an LSN. If this were not true Rose would have to add another column to the basic record table. In addition to an LSN it would need the offset into the extended record log for basic records which refer to application-specific provenance. Compressing the basic record table with this additional column would be more difficult and would increase Story Book's overhead. Records that have no application-specific data to store need not incur an additional write since the magic number for an empty extended record is zero and can be recorded by making a hole in the file. Reading the extended record need only be done when explicitly checking for a particular kind of application-specific provenance in a basic record.

3.7 Provenance Query

Story Book acquires the provenance of a file by looking up its inode in the (`file name`, `ino`) hash table. The inode is used to begin a breadth first search in the basic and process record tables. We perform a transitive closure over sets of processes that wrote to the given file (or other files encountered during the search), over the set of files read by these processes, and over the parents of these processes. We search in order of decreasing LSN and return results in reverse chronological order.

3.8 Compression and Record Suppression

Rose tables provide high-performance column-wise data compression. Rose's basic record table run length encodes all basic record columns except the LSN, which is stored as a 16 bit diff in the common case, and the opcode, which is stored as an uncompressed 8 bit value. The process record table applies run length encoding to the process id, and ex#. It attempts to store the remaining columns as 16 bit deltas against the first entry on the page. The hash tables do not support compression, but are relatively small, as they only store one value per executable name and file in the system.

In addition to compression, Story Book supports *record suppression*, which ignores multiple calls to read and write from the same process to the same file. This is important for programs that deal with large files; without record suppression, the number of (compressed) entries stored by Rose is proportional to file sizes.

Processes that wish to use record suppression need only store the first and last read and write to each file they access. This guarantees that the provenance graphs generated by Story Book's queries contain all tainting processes and files, but it loses some information regarding the number of times each dependency was established at runtime. Unlike existing approaches to record suppression, this scheme cannot create cyclic dependency graphs. Therefore, Story Book dependency graphs can never contain cycles, no special cycle detection or removal techniques are required.

However, record suppression can force queries to treat suppressed operations as though they happened in race. If this is an issue, then record suppression can be disabled or (in some circumstances) implemented using application-specific techniques that cannot lead to the detection of false races.

Note that this scheme never increases the number of records generated by a process, as we can interpret sequences such as "open, read, close" to mean a single read. Record suppression complicates recovery slightly. If the sequence "open, read, ..., crash" is encountered, it must be interpreted as: "open, begin read, ..., end read, crash." Although this scheme requires up to two entries per file accessed by each process, run length encoding usually ensures that the second entry compresses well.

4 Evaluation

Story Book must provide fast, high-throughput writes in order to avoid impacting operating system and existing application performance. However, it must also index its provenance records so that it can provide reasonable query performance. Therefore, we focus on Story Book write throughput in Section 4.2, a traditional file system workload in Section 4.3, and a traditional database workload in Section 4.4. We measure provenance read performance in Section 4.5.

4.1 Experimental Setup

We used eight identical machines, each with a 2.8GHz Xeon CPU and 1GB of RAM for benchmarking. Each machine was equipped with six Maxtor DiamondMax 10 7,200 RPM 250GB SATA disks and ran CentOS 5.2 with the latest updates as of September 6, 2008. For the file system workload in Section 4.3 Story Book uses a modified 2.6.25 kernel to ensure proper write ordering, and for all other systems an unmodified 2.6.18 kernel was used. To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly formatted file system with as few services running as possible. For query benchmarks, database files are copied fresh to the file system each time. We ran all tests at least four times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student's-t distribution. In each case, unless otherwise noted, the half widths of the intervals were less than 5% of the mean. Wait time is elapsed time less system and user time and mostly measures time performing I/O, though it can also be affected by process scheduling. Except in Figure 6 and Figure 7 where we are measuring throughput and varying C_0 size explicitly, all cache sizes are consistent for all systems across all experiments. Berkeley DB's cache is set to 128MiB, and Stasis' page cache is set to 100MiB with a Rose C_0 of 22MiB, allocating a total of 122MiB to Fable.

Provenance Workloads. The workload we use to evaluate our throughput is a shell script which creates a file hierarchy with an arity of 14 that is five levels

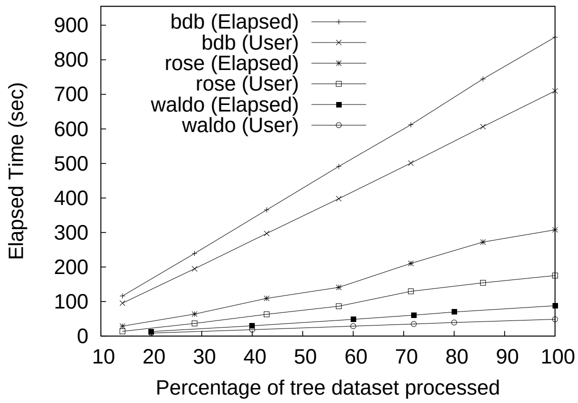

Figure 6: Performance of Waldo and Story Book's insert tool on logs of different sizes.

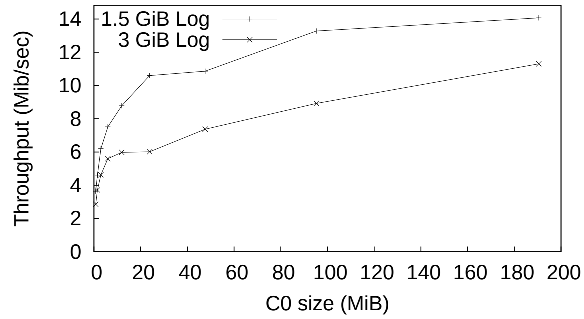

Figure 7: Performance of Story Book's Fable backend with varying in-memory tree component sizes.

deep where each directory contains a single file. Separate processes copy files in blocks from parent nodes to child nodes. This script was designed to generate a dense provenance graph to provide a lower bound on query performance and to demonstrate write performance. We refer to this script as the *tree dataset*. With no WAL truncation, the script generates a 3.5GiB log of uncompressed provenance data in Story Book, and a 244MiB log of compressed (pruned) provenance data in PASSv2. Fable compresses provenance events on the fly, so that the database file does not exceed 400MiB in size.

4.2 Provenance Throughput

PASSv2 consists of two primary components: (1) an in-kernel set of hooks that log provenance to a write-ahead log, and (2) a user-level daemon called *Waldo* which reads records from the write-ahead log and stores them in a Berkeley Database (BDB) database. We used a pre-release version of PASSv2, which could not reliably complete I/O intensive benchmarks. We found that their pre-release kernel was approximately 3x slower than Story Book's FUSE-based provenance file system, but expect their kernel's performance to change significantly as they stabilize their implementation. Therefore, we focus on log processing comparisons here.

For both Story Book and PASSv2, log generation was at least an order of magnitude slower than log processing. However, as we will see below, most of Story Book's overhead is due to FUSE instrumentation, not provenance handling. We expect log processing throughput to be Story Book's primary bottleneck when it is used to track provenance in systems with lower instrumentation overheads.

By default, Waldo durably commits each insertion into BDB. When performing durable inserts, Waldo is at least ten times slower than Fable. Under the assumption that it will eventually be possible for PASSv2 to safely avoid these durable commits, we set the `BDB_TXN_NOSYNC` option, which allows BDB to commit without performing synchronous disk writes.

We processed the provenance events generated by the tree dataset with Waldo and Story Book. This produced a 3.5GiB Story Book log and a 244MiB Waldo log. We then measured the amount of time it took to process these logs using Waldo, Fable and Story Book's Berkeley DB backend.

Rose uses 100MiB for its page cache and 172MiB for its C_0 tree. Both Berkeley DB and Waldo use 272MiB for their page cache size. Our optimized version of Waldo performs 3.5 times faster than Fable. However, PASSv2 performs a significant amount of preprocessing, record suppression and graph-pruning to its provenance data before writing it to the Waldo log. Fable interleaves these optimizations with provenance event handling and database operations. We are unable to measure the overhead of PASSv2's preprocessing steps, making it difficult to compare Waldo and Fable performance.

Story Book's Berkeley DB backend is 2.8 times slower than Fable. This is largely due to Fable's amortization of in-memory buffer manager requests, as User CPU time dominated Berkeley DB's performance during this test. Because Rose's LSM-trees are optimized for extremely large data sets, we expect the performance gap between Berkeley DB and Fable to increase with larger provenance workloads.

4.3 File System Performance

Story Book's write performance depends directly on the size of its in-memory C_0 tree. In Figure 7 we compare the throughput of our Rose backend when processing a 1536MiB WAL and 3072MiB WAL generated from our tree dataset. Since Rose must perform tree merges each time C_0 becomes full, throughput drops significantly if we allocate less than 22MiB to C_0. We set C_0 to 22MiB for the remainder of the tests.

Figure 8 measures the performance of a multi-

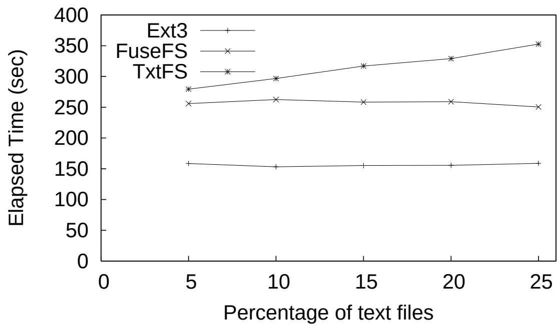

Figure 8: Performance of Story Book compared to `ext3` and FUSE for I/O intensive file system workloads.

threaded version of Postmark [5] on top of `ext3`, on top of `FuseFS`, a "pass through" FUSE file system that measures Story Book's instrumentation overhead, and on top of `TxtFS`, a Story Book provenance file system that records diffs when `.txt` files are modified. When running under `TxtFS`, each modification to a text file causes it to be copied to an object store in the root partition. When the file is closed, a diff is computed and stored as an external provenance record. This significantly impacts performance as the number of `.txt` files increases.

We initialized Postmark's number of directories to 890, its number of files to 9,000, its number of transactions to 3,600, its average filesize to 190 KiB, and its read and write size to 28KiB. These numbers are based on the arithmetic mean of file system size from Agarwal's 5-year study[1]. We scaled Agarwal's numbers down by 10 as they represented an entire set of file system modifications, but the consistent scaling across all values maintains their ratios with respect to each other. We ran this test using 30 threads as postmark was intended to emulate a mail server which currently would use worker threads to do work. Because FUSE aggregates message passing across threads, both `TxtFS` and `FuseFS` perform better under concurrent workloads with respect to `Ext3`. We repeated each run, increasing the percentage of `.txt` files in the working set from 5% to 25%, while keeping all the other parameters the same (Text files are the same size as the other files in this test).

`FuseFS` and `ext3` remain constant throughout the benchmark. `TxtFS`'s initial overhead on top of `FuseFS` is 9%, and is 76% on top of `Ext3`. As the number of copies to `TxtFS`'s object store and the number of diffs increases, `TxtFS`'s performance worsens. At 25% `.txt` files, `TxtFS` has a 40% overhead on top of `FuseFS`, and is 2.2 times slower than `Ext3`.

	Regular Response Time (s)	**Story Book Response Time (s)**
Delivery	52.6352	53.423
New Order	45.8146	48.037
Order Status	44.4822	47.4768
Payment	44.2592	46.6032
Stock Level	53.6332	55.1508
Throughput (tpmC)	201.82	196.176

Table 1: Average response time for TPC-C Benchmark

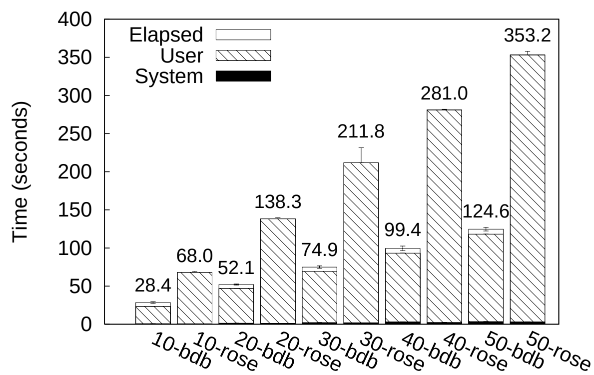

Figure 9: Performance of a provenance query.

4.4 MySQL TPC-C

Next, we instrumented MySQL in order to measure the performance of a user-level Story Book provenance system. This workload has significantly lower instrumentation overhead than our FUSE file system. We implemented a tool that reads MySQL's general query log and determines which transactions read from and wrote to which tables. It translates this information into Story Book provenance records by treating each operation in the general log as a process (We set the process "executable name" to the query text).

We measured the performance impact of our provenance tracking system on a copy of TPC-C configured for 50 warehouses and 20 clients. In Table 1 we list two configurations: `Regular` and `Story Book`. Response times under Story Book increased by an average of 4.24%, while total throughput dropped by 2.80%.

4.5 Provenance Read

To evaluate Story Book's query performance, we compared the query performance of our Berkeley DB backend to that of Fable. Our BDB database uses the same table schema and indexes as Fable, except that it uses B-trees in the place of Stasis' Rose indexes.

We generated our database by processing the entire tree dataset, which is designed to create dense provenance graphs. This results in a 400MiB database for

Fable, and a 905MiB database for Berkeley DB. We randomly selected a sequence of 50 files from our tree dataset, and we perform just the first 10 queries in our first data point, then the first 20 queries and so on.

In both cases, the data set fits in RAM, so both systems spend the majority of their time in User. As expected, runtime increases linearly with the number of queries for both systems. Fable is 3.8 times slower than Berkeley DB in this experiment.

Provenance queries on the tree dataset result in a large number of point queries because our database is indexed by inode, and each inode only has a few entries. Although Rose provides very fast tree iterators, it must check multiple tree components per point query. In contrast, Berkeley DB must only consult a single B-tree per index lookup. Rose was originally intended for use in versioning systems, where queries against recently modified data can be serviced by looking at a single tree component. Our provenance queries must obtain all index entries, and cannot make use of this optimization.

5 Conclusions

Our implementation of Story Book focuses on minimizing the amount of state held in RAM and on simplifying the design of provenance systems. It is able to either record exact provenance information, or to apply a straightforward record-suppression scheme. Although our Berkeley DB based backend provides superior query performance, the Fable backend provides superior write throughput, and should scale gracefully to extremely large data sets.

Story Book's support for system-specific durability protocols allows us to apply Story Book to a wide range of applications, including MySQL. It also allows us to use Valor to avoid synchronous log writes, removing the primary bottleneck of existing systems.

Story Book's application development model and the simplicity of our provenance data model make it easy to add support for new types of provenance systems, and to store additional information on a per-file basis.

Story Book's primary overhead comes from its FUSE file system instrumentation layer. Our experiments show that Story Book's underlying provenance database provides much greater throughput than necessary for most file system provenance workloads. This, coupled with Story Book's extensibility, allow it to be applied in more demanding environments such as databases and other multiuser applications.

Acknowledgments. We would like to thank the reviewers for their comments and advice. We would also like to thank Leif Walsh and Karthikeyan Srinaivasan for their extensive help in running and analyzing experiments for the camera ready copy.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST '07: Proc. of the 5th USENIX conference on File and Storage Technologies*, pp. 3–3, Berkeley, CA, USA, 2007.

[2] Dublin Core Metadata Initiative. Dublin core. dublincore.org, 2008.

[3] J. Clark et al. ExPat. `expat.sf.net`, Dec 2008.

[4] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, 2007.

[5] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[6] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul. 1978.

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

[8] D. Morozhnikov. FUSE ISO File System, Jan. 2006. `http://fuse.sf.net/wiki/index.php/FuseIso`.

[9] K. Muniswamy-Reddy, J. Barillari, U. Braun, D. A. Holland, D. Maclean, M. Seltzer, and S. D. Holland. Layering in provenance-aware storage systems. Technical Report TR-04-08, Harvard University Computer Science, 2008.

[10] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pp. 1–14, Seattle, WA, Nov. 2006.

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

[12] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[13] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proc. of the VLDB Endowment*, volume 1, Auckland, New Zealand, 2008.

[14] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, Sept. 2005.

[15] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27 edition, Dec. 2004. `www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html`.

[16] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proc. of the seventh usenix conference on file and storage technologies*, San Francisco, CA, Feb. 2009.

[17] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok. Rapid file system development using ptrace. In *Proc. of the Workshop on Experimental Computer Science , in conjunction with ACM FCRC*, page Article No. 22, San Diego, CA, Jun. 2007.

[18] Sun Microsystems. MySQL. `www.mysql.com`, Dec 2008.

[19] W3C. Resource description framework. w3.org/RDF, 2008.

[20] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of the 2005 CIDR Conf.*, 2005.