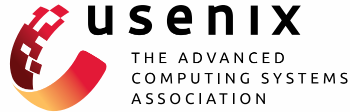

A Modular and Efficient Past State System for Berkeley DB

Ross Shaull, ***NuoDB;*** **Liuba Shrira,** ***Brandeis University;***
Barbara Liskov, ***MIT/CSAIL***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/shaull

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

A Modular and Efficient Past State System for Berkeley DB *

Ross Shaull
NuoDB

Liuba Shrira
Brandeis University

Barbara Liskov
MIT/CSAIL

Abstract

Applications often need to analyze past states to predict trends and support audits. Adding efficient and non-disruptive support for consistent past-state analysis requires after-the-fact modification of the data store, a significant challenge for today's systems. This paper describes Retro, a new system for supporting consistent past state analysis in Berkeley DB. The key novelty of Retro is an efficient yet simple and robust implementation method, imposing 4% worst-case overhead. Unlike prior approaches, Retro protocols, backed by a formal specification, extend standard transaction protocols in a modular way, requiring minimal data store modification (about 250 lines of BDB code).

1 Introduction

Applications need *retrospection*, the ability to analyze past states, to provide audits and predict trends. Without adequate support in the data store, it is hard for developers to reconstruct consistent past states corresponding to events of interest. Yet, many data stores lack support for retrospection because adding a low-impact consistent past state system to a data store has been challenging using current approaches.

This paper describes Retro, a system that adds retrospection to Berkeley DB (BDB), a popular transactional key-value database. Our system automatically stores past states of interest to the application, and allows the applications to query automatically restored consistent past states. The queries can take advantage of the application code base; any read-only application or library program that runs in BDB can also run retrospectively.

The novel contribution of Retro is an efficient yet simple and robust implementation method. Retro is very efficient; it does not disrupt BDB performance even when applications retain the past states at high frequency, imposing a minimal 4% performance penalty in the worst case. Furthermore, Retro extends standard database protocols in a modular and robust way, based on a simple past state system specification that serves as a basis for a formal proof of Retro protocol correctness (presented in [13]); this is in contrast to prior past state systems (e.g., [5, 15, 17]), which used more complex and fragile ad-hoc modifications to data store internals to avoid disrupting database performance. The code implementing Retro protocols composes with standard interfaces in the database software stack. Because the composition is modular, the modifications to the database code are minimal: our prototype modifies only 250 lines of Berkeley DB source code.

To preserve transaction performance when saving consistent past states, Retro captures the needed past states incrementally, using split copy-on-write [15]. It then creates the past state lookup structures [14] and accumulates the past states and lookup structures in additional memory that is dedicated to storing past state information. Retro writes the past states and lookup structures to a separate log-structured store, in the background without interfering with database queries. Retro recovery ensures the past states and lookup structures remain consistent and durable in the presence of crashes. To query past states, Retro redirects code to snapshot pages using dynamic translation structures, without interfering with database transactions.

Retro accomplishes its tasks by extending BDB update commit, update recovery, and update writing and reading protocols in a simple and intuitive way. An extension to the commit protocol retains the needed past states when an update transaction commits. Similarly, an extension to the BDB recovery protocol retains the needed past states when BDB recovers updates from the transaction log after a crash, thus relegating past state recovery to database recovery, an important simplification since recovery pro-

[1]This work was partially supported by the National Science Foundation under grants NSF IIS-1251037, NSF CNS-1318798. The work was accomplished when Ross Shaull was at Brandeis University.

tocols can be complex.

Retro recovery faces two complications. Care must be taken to ensure that BDB does not discard the transaction log before past states become durable, and to ensure that recovery attempts that fail and restart do not corrupt past states. These dangers are avoided by enforcing a simple invariant ensuring that past states become durable before database updates.

Retro is implemented as a system of concurrent callbacks running as part of BDB commit, recovery, and writing and reading protocols. The concurrent callbacks access shared state, e.g., to track which past states have been already saved. The callbacks must therefore run in a thread safe manner, and moreover must be serialized in a consistent order to ensure consistent past states can be identified correctly. If Retro callbacks were to block unnecessarily, this could increase BDB transaction latency. To avoid this bottleneck, Retro stores its shared state in specialized data structures that eliminate blocking.

In summary, the contributions of this paper include 1) a new efficient system for retrospection in Berkeley DB, implemented using a simple and robust method, avoiding invasive database modifications. 2) modular past state protocols justified by a formal specification that extend standard transaction protocols, 3) design of the modular snapshot layer that implements the past state protocols, using specialized data structures that minimize overheads to BDB, 4) experimental evaluation supporting our efficiency claims, and analysis of retrospection performance for in-memory and on-disk past states.

Retro was designed for BDB but we believe our method is general and can be applied to other transactional data stores, contributing a step towards making past state support more widely available to applications.

2 Related Work

Past states can be supported at the level of logical records or files (e.g., [5, 12]), or low-level pages (e.g. [17]), like Retro. Without close integration with the data store, past state systems can impose a high performance penalty to provide transactionally consistent records, or crash consistent files. Systems that operate *below* the data store (e.g., page-level Windows VSS), block update transactions, disrupting performance if snapshots are frequent. Temporal databases that operate *above* a database, restrict scalability [9].

Integrated past state systems can exploit data store mechanisms to write consistent past states at low cost. Postgres [3] and versioning file systems (e.g., [4]) integrate with a no-overwrite system that keeps past records in place, and copies new records to a new place, reducing the number of writes. Since past and current state are not separated, large past state can negatively impact the cost of current state reads [3]. Read impact can be avoided by keeping the past state separate. Ganymed [10], a Postgres based system, copies past records to a separate Postgres replica node, using replication middleware. The replica provide access to historical snapshots using modified concurrency control and query protocols. Read impact can also be avoided by exploiting recovery (e.g. fuzzy checkpoints [3]), but recovery based methods are too slow for on-line programs.

ImmortalDB [5] supports consistent past records, integrating with SQL Server. The database data layout is modified to keep recent versions of past records on the current state pages, eventually migrating old versions to separate pages; indexes are modified to support temporal access. Oracle Total Recall supports integrated historical record tables, indexed like regular tables. SNAP [15] supports consistent indexed split page-level snapshots, integrating with an object store. SFS [17], supports split page-level snapshots, integrating with a file system. All above systems integrate past state support using invasive modifications to the data store internals. VersionFS [7] adds versioning in a stackable file system. The stackable architecture supports modular extension, a goal shared by Retro .

Retro adopts the split snapshot representation in SNAP [15] and Skippy index [14], extending the prior work in important ways. The Skippy work provides efficient multi-level index for split snapshots, without considering index recovery, concurrency, or implementation of a complete snapshot system. Retro implements a complete snapshot system, including efficient recovery and non-blocking concurrency control protocols for Skippy index and snapshot data. Unlike SNAP, and other implementations using invasive ad-hoc modifications, Retro provides an efficient implementation method based on modular extension of standard transaction protocols. The modular snapshot protocols, based on a formal snapshot system model, and their low-cost implementation structures in BDB are the new contributions of Retro.

3 Programming Model

From the application developer's perspective, programming with Retro is a straightforward extension to programming with BDB using a simple *named* snapshot abstraction (Figure 1).

Retro supports C and SQL (SQLite) BDB APIs. Applications run transactions that issue update and query requests to records organized in tables. An applications may declare a persistent snapshot at transaction boundary at any time by issuing the *snapshot now* command. The declaration command commits a transaction, whose serialization point defines the contents of the snapshot. A snapshot represents the state of the entire database (e.g.,

Current-state queries are unchanged by Retro
results ← select * from ...
Applications may declare snapshots at any time and get back a snapshot identifier
S ← snapshot now
As of queries are delimited with snapshot identifier
results ← as of **snapshot S** { select * from ... }

Figure 1: Programming with snapshots

tables, indexes, system catalogs).

After declaration commits, the application is returned a *snapshot identifier* which permanently identifies the declared snapshot. A snapshot identifier may be used to access a snapshot immediately; no delay is required between declaring and accessing a snapshot. Retro does not mandate how snapshot identifiers are remembered for later use, e.g. they can be stored in a table along with a timestamp. Internally, Retro assigns to snapshot identifiers consecutive integer names in declaration commit order.

To make use of the snapshots they declare, applications run queries as of those snapshots. The application delimits any *read-only* query code with *as of* to instruct Retro to run the delimited query retrospectively. A query delimited by *as of snapshot S* reflects the effects of all the transactions committed prior to the serialization point of *S*, and none of the effects of transactions committed after *S*. Inside the *as of* delimiter, the query itself is written just like a normal, current-state query. This makes it easy to leverage existing programmer knowledge and codebases when programming with Retro.

Existing current-state BDB code runs unmodified. Code that declares snapshots and runs retrospection can be executed alongside the current-state code in the same database, making it easy to use retrospection from current state BDB code on-line where needed.

4 Retro architecture

The relevant components of BDB software stack are depicted in Figure 2. The shaded areas show Retro extensions (explained later). A BDB application runs transactions that manipulate logical data records and tables, issuing update and query requests against the BDB API, which processes the requests and translates them into requests to the transactional *storage manager*. The storage manager manages logical data records organized in pages, stored on durable storage. The pages are the unit of transfer between durable storage (on disk) and the page cache (in memory).

When an application requests a data record as part of some query (e.g., “get record k”), that request is pro-

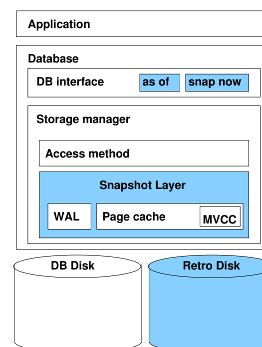

Figure 2: Retro modularized in database architecture

cessed by an *access method* (a binary tree or a hash table), a storage manger component that encapsulates the representation of records in a table. The access method translates requests to read and write data records into requests to the page cache to read and write data in units of pages. Pages are cached in the page cache in memory, and data records are read from and written to the cached pages.

A request for a page issued to the page cache by an access method refers to the page by its *logical name*. A logical name is a pair $(file, number)$, where file is the identifier of a database file open in the page cache, and number is some offset within that file. The logical page name gets translated to its physical disk address when page cache performs disk I/O.

The storage manager includes two additional component relevant to Retro, concurrency control and recovery, responsible for transaction serialization and crash consistency, respectively. These components will be described later when we explain how Retro extends them.

4.1 BDB components extended by Retro

The mechanisms that implement Retro functionality are modularized within the *snapshot layer* (SL) (figure 2). The SL wraps components in the transactional storage manager, extending BDB behavior to add efficient snapshot creation and querying. Snapshots are created and accessed by extending the page cache and concurrency control (section 4.4 and section 5). Snapshot recovery supporting efficient snapshot creation is achieved by extending recovery (section 6).

Retro does not affect access methods, the logical-to-physical translation of page names into their disk addresses, or how the database is organized on disk. These storage manger components are transparent to Retro. Retro is concerned with logical page names, and expanding that namespace to support snapshots.

Retro assumes that page cache memory and extra

storage is allocated for holding snapshots. The extra page cache memory allows the system to accumulate snapshots before writing them to disk without contending with current state transactions. For best performance, the extra storage is on separate disks from the database, to avoid interference. Snapshots have the potential to consume much more storage that the current state, since (generally speaking) history grows with any insert, update, or delete, while the current state only grows when data is inserted. Retro helps use an investment in additional storage efficiently by being selective (Retro keeps only declared snapshots) and incremental (snapshots may share snapshot pages). Currently, Retro does not allow snapshot deletion. Adopting low-cost snapshot deletion [16] is a straightforward extension.

4.2 Low-level snapshots

At the level of the storage manager, a snapshot S is a *complete* and *transactionally consistent* collection of *snapshot pages*. The collection is complete because it contains all the data and metadata pages in BDB, including index and catalog pages. The collection is transactionally consistent because the snapshot pages reflect the serialization point of the snapshot declaration command, making snapshot pages consistent with one another. These properties allow any new or existing read-only code that could run in the database when snapshot S was declared, to run *as of* a snapshot S.

The page-level Retro snapshot abstraction makes it possible to run retrospective queries without changing access methods, indices, and other storage system code that relies on page layout and naming. The same name used to denote a page in the current state is used during retrospection to denote a snapshot version of that page. This means that snapshot pages that refer to each other by name (e.g., index pages) can be used normally by storage system code.

For convenience, we denote the version of a page P as of a snapshot S using $P@S$. This is purely notational; when BDB requests a page P while querying restrospectively as of S, it requests P using the same page name as it would if it were requesting P as part of a current-state query. Section 4.4 describes this mechanism in detail.

4.3 Snapshot overwrite sequence

Retro creates snapshots using split copy-on-write [15] and Skippy index [14]. When a snapshot is declared, all snapshot pages are *shared* with the current database pages. When a BDB transaction updates a page that is *shared* with a snapshot, Retro saves the snapshot page in memory, updates the snapshot indexing metadata, and eventually writes pages and metadata to the Retro disk.

We use the notion of *Snapshot Overwrite Sequence* (OWS) to reason about what Retro protocols need to do, i.e. as a correctness specification.

> **Definition**: Let H be a serial committed transaction execution history. The snapshot *overwrite sequence* of H (OWS(H)) is a mark up of H that tags every snapshot declaration and every commit that updates a page *shared* with a snapshot.

OWS(H) captures the points in the execution sequence where snapshot pages and index metadata must be created. There is a straightforward formal proof of correctness of Retro protocols based on OWS [13], omitted for lack of space. OWS answers the following questions: **1) which page versions to save**: the page pre-state of the first update to a page following a snapshot declaration (Sec 5), **2) what state to recover after a crash**: a recovery after a crash immediately following execution H must recover all pages and metadata created by operations tagged in OWS(H) (Sec 6), **3) where to get the page P@S requested by a retrospective query running after H**: from Retro, if there is a tagged commit updating P following the declaration of S in H, or otherwise from the database (Sec 7).

For example, consider the following OWS(H):

$$T1(S1)T2(wR)T3(S2)T4(wRwQ)$$
$$T5(wRwQ)T6(S3)T7(wRwP)$$

Retro saves pre-states in $T2$ (R), $T4$ (R,Q), and $T7$ (R,P) but not $T5$ since its updates to R and Q are not the first modifications to a page following the declaration of snapshot $S2$. For a crash following H, Retro recovers all the pre-states and indexing state created in $T1$ to $T4$, $T6$ and $T7$. A retrospective query as off snapshot 3, when issued after H, gets $P@3$ from Retro, but when issued before $T7$, gets $P@3$ from the database.

4.4 Logical page virtualization

Retro allows retrospective queries to execute concurrently with current state queries and updates in the same page cache, allowing current state programs to directly query past states. The key idea behind the execution architecture for retrospection is to translate the BDB logical page names to the names of snapshot pages when a query runs retrospectively, using logical page virtualization in the snapshot layer (SL). The name translation is transparent to BDB, enabling storage system code using logical page names to run on both the current state and snapshots.

Figure 3 depicts the retrospective query execution path. Retro stores the snapshot pages in a file called

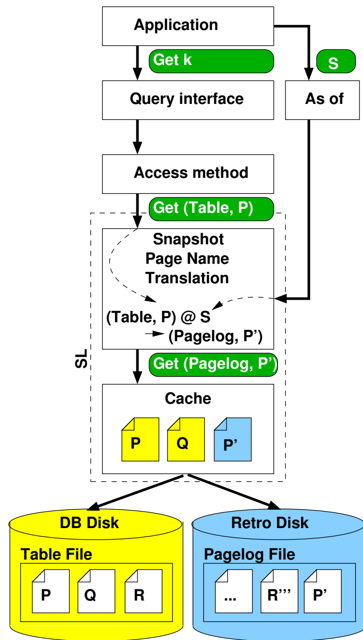

Figure 3: Retrospective query execution

Pagelog located on the Retro disk. This file is opened in the page cache like any other database file. A snapshot page $P@S$ that has been overwritten since S was declared will be copied to Pagelog. For brevity, we refer to the offset in Pagelog where $P@S$ is stored as P'; the actual offset P' has no relationship to the logical name of P in the current state.

The logical page virtualization is implemented by a *translation component*. The translation component intercepts page requests from access methods and translates logical database page names into logical snapshot page names if the code issuing the request is running retrospectively. The *as of* primitive provided by the Retro interface extension identifies the snapshot from which a requested page should be read.

The translation component keeps track of translations from logical names (e.g., page P in a database file $Table$) to pre-states in Pagelog (e.g., P' in $Pagelog$) for any declared snapshot. After translating $(Table,P)@S$ to $(Pagelog,P')$, the translated name is passed to the page cache, which reads and caches it like any other page. The contents of the snapshot page $(Pagelog,P')$ are returned to the access method as though it were the current-state contents of the page named $(Table,P)$, with the access method and application none the wiser that the requested page was transparently switched with a snapshot page. If the retrospectively accessed page named $(Table,P)$ is still shared with the database, the name translation in SL will be identity function, and the logical name $(Table,P)$ will be requested from the cache.

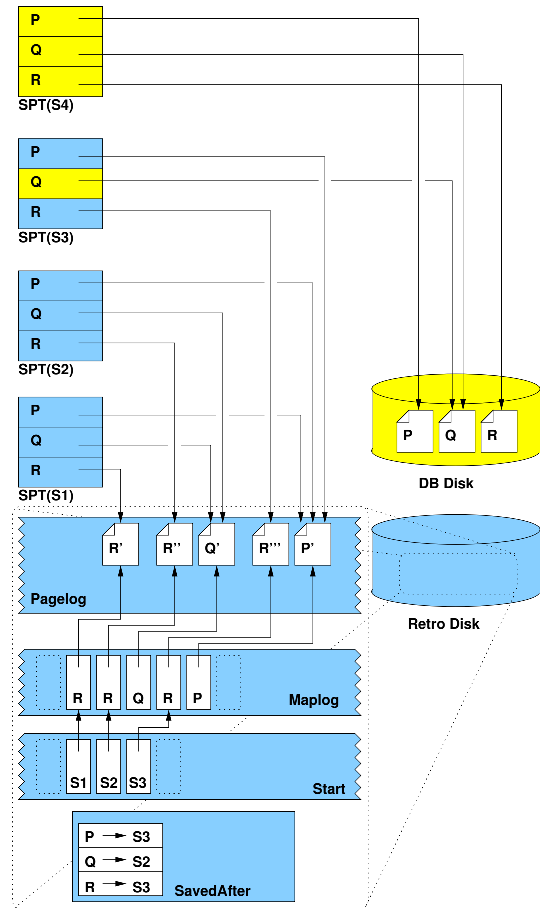

Figure 4: Snapshot representation

BDB programs that read and write the current state are unaffected by Retro. When the application performs a query without specifying *as of*, Retro does not engage page name translation, and the logical page name is passed unchanged to the page cache. Because different versions of the same page have distinct logical names, they can coexist in the cache.

4.5 Retro snapshot representation

Retro adapts the *split snapshot* representation [15, 16] to organize snapshot data (pre-states) and metadata (indices for page name translation) . Split snapshots are created at the page level, and stored separately ("split") from the database (e.g., on a separate disk). By splitting snapshots from the database, the approach does not affect how the database organizes data, and partially isolates database I/O performance from snapshot-related I/O.

Figure 4 depicts the organization of Retro data and metadata. When an update U tagged in the OWS commits a modification to a page P, Retro saves in memory the the *pre-state* of P created by U. For every saved pre-

state, Retro also updates snapshot metadata in memory to note the snapshots to which the pre-state of P belongs. Snapshot metadata includes *Maplog*, which maps from logical page names *as of* a declared snapshot to the location of pre-states in Pagelog; *SavedAfter*, which tells if a snapshot shares a particular page with the current state by tracking for every database page the latest snapshot for which a pre-state of that page was saved; and *Start*, which associates every declared snapshot with the first Maplog entry created for that snapshot.

Snapshot data and metadata are stored on disk. The on-disk representation in Figure 4 corresponds to the example transaction history in Sec 4.3. Section 6 explains how Retro writes snapshot data and metadata to disk in efficient and recoverable manner.

Snapshot Page Tables Snapshot page tables (SPTs) are in-memory tables used to implement snapshot page translation during retrospection (section 4.4). SPT entries map the logical page names in the database to the logical names of snapshot pages that can be either pre-states in Pagelog, or pages shared with the database. Resolving the snapshot page name $P@S$ to the logical name of a page in Pagelog requires looking up P in $\mathrm{SPT}(S)$. In Figure 4, $\mathrm{SPT}(S1)$ and $\mathrm{SPT}(S2)$ share Q' since there was no update to Q between S1 and S2. In $\mathrm{SPT}(S3)$, Q points to the database because Q has yet been modified since $S3$ was declared.

Maplog, Start, and SavedAfter Keeping SPTs in memory for every declared snapshot would be costly, so instead Retro reconstructs SPTs from the saved metadata. When a retrospective query is run as of S, Retro builds $\mathrm{SPT}(S)$ by scanning Maplog for the first occurrence of a mapping for each page; these first-encountered mappings correspond to the pre-states saved from update tagged in the OWS. Retro scans Maplog from Start (S) since earlier mappings correspond to pre-states saved before S was declared. A scan will not encounter mappings for snapshot pages that are still shared with the database. Retro can determine this, without scanning to the end of Maplog, from the data structure SavedAfter.

E.g., in figure 4, SavedAfter shows pages P and R were last saved after $S3$ was declared, but the snapshot after which Q was last saved is $S2$.

Naively scanning Maplog can be expensive, because mappings for pages for which pre-states are frequently saved (due to update skew) increase the length of a Maplog scan. An efficient indexing technique to combat the impact of update skew called Skippy, is described in [14].

An SPT built for a retrospective query Q running as of snapshot S needs to be kept up to date as Retro saves snapshot pages. The techniques used to accomplish this are discussed in [13].

5 Extending MVCC

BDB serializes transactions using a popular multi-version page-level concurrency control protocol (MVCC) that enables concurrent transaction reads and updates by keeping multiple versions of pages in memory. Every transaction T "sees" a consistent view of the database, called an isolation snapshot, consisting of the versions of pages which were most-recently committed *before* T began. If T updates a page, the new page version becomes visible to other transaction after T commits. MVCC versions pages in the page cache [1, 8] in the storage manager.

Retro extends MVCC, creating persistent snapshots out of isolation snapshots. To avoid confusion, we refer to the persistent snapshots created by Retro as *Psnaps* and the volatile isolation snapshots created by MVCC as *Vsnaps*.

MVCC mechanisms When an application requests a page P for *update* (via an access method) as part of transaction T, a private copy of P is created in the cache and associated with T. When T commits, P is marked with a number called the *commit LSN* of T (Sec 6), identifying the last transaction that updated P. The versions of P are linked together in a list called the *version chain*.

When an application requests to *read* a page P (via an access method) as part of transaction T, the page cache searches the version chain for the latest version committed before T began. If the page cache has no versions of a page P, P is read from the disk. To avoid overflowing the cache, MVCC garbage collects page versions on each version chain that are no longer visible to any running transactions. Every transaction has an associated Vsnap, and Vsnaps of multiple transactions may coexist in the same cache.

5.1 Persisting snapshots

To persist a Psnap S, Retro needs to save the latest version of every page committed before S was declared, and eventually write it to Pagelog on the Retro disk.

Retro saves pages from S using page-level copy-on-write in the page cache. It relies on MVCC to create, in memory, the needed page copies in Vsnaps of transactions that update pages following a snapshot declaration, and uses SavedAfter data structure to identify the needed versions among these copies. SavedAfter relates every logical page name to the latest snapshot after which a

version of it was last saved. When a transaction T commits, Retro checks SavedAfter to see if any page P updated by T is the first update to be committed since the latest-declared snapshot S, and if so, saves the pre-state of P created by T and updates SavedAfter.

Retro writes the saved pre-states efficiently using asynchronous I/O. However, pre-states which are no longer visible in any Vsnap are candidates for MVCC garbage collection. So, Retro must maintain the following invariant to ensure that pre-states needed for a declared Psnap can be written to Pagelog:

> **Before-GC invariant**: any pre-state from an update in the OWS is saved for Pagelog before MVCC garbage collects it.

Retro maintains this invariant by writing the pre-states in the Vsnap of a transaction T lazily after T commits, postponing garbage collection of the pre-states for a short time if necessary. Since MVCC garbage collection also happens lazily when page replacement is needed, the delay has minimal impact.

6 Extending Recovery

Snapshot recovery is greatly simplified by leveraging the native BDB recovery mechanism. When BDB recovery replays the updates of transactions that committed before the crash, Retro saves the intermittent page versions that correspond to the pre-states of updates tagged in OWS, mimicking snapshot creation during normal execution. This way, if BDB creates intermittent page versions identical to the ones created during normal execution, when recovering a history H, Retro will create snapshot pages and metadata defined by OWS(H).

During recovery, BDB produces the same intermediate versions that were produced by the earlier execution as long as the page versions read from disk after the crash are the same versions used by original updates. Care must be taken to ensure that snapshots are correctly recovered. BDB may discard the update records from the log after updates are written to the database. Shapshots could be lost if updates records are discarded prematurely. Moreover, if BDB recovery crashes while updating the database pages on disk, then when recovery restarts, it may encounter disk page versions different from those needed by OWS, causing snapshot pages to be lost.

Retro avoids the complications by enforcing the following simple write-ordering invariant, called the *write-ahead snapshot invariant*, during normal operation and during recovery:

> **Write-ahead snapshot (WAS) invariant**: The pre-state of P (and associated snapshot metadata) from an update in $\mathrm{OWS}(H)$ must be written to the Retro disk before the version of P created by that or any later update in H is written to the database disk.

WAS invariant guarantees snapshot pages and metadata become durable before database pages needed to recover them become overwritten.

During replay (like in normal execution), Retro consults snapshot metadata e.g., SavedAfter, to check if the pre-state needs to be saved. Snapshot metadata therefore, must be recovered the database. Retro simplifies metadata recovery by storing all metadata (Start, Maplog, and SavedAfter) in BDB transactional data structures, and updating metadata using regular transactions, Retro relies on BDB to recover metadata.

It would be costly to write snapshot data using a database transaction, since it is large compared to snapshot metadata. Instead snapshot data is written using regular writes. So, the Retro recovery protocol enforces a second write-ordering invariant to make it possible to clean up partially-written snapshot data after a crash and correctly detect whether a particular pre-state was written to the Retro disk before the crash:

> **Snapshot-data-before-metadata invariant**: Before Retro commits the mapping for pre-state P', it writes P' to Pagelog.

The Retro write invariants, enforced during normal operation and recovery, and the two stages of Retro recovery (metadata and replay), guarantee that for every BDB recovery of transaction history H, Retro will correctly recover snapshot data and metadata defined by OWS(H), even in the presence of repeated BDB recoveries.

6.1 BDB recovery

BDB follows the write-ahead logging (WAL) protocol to ensure recoverability after a crash. The WAL protocol requires that the database write a record of updates made by a transaction T in a durable log before T commits. Each update is represented by a log record which contains (at least) the information required to repeat the update; this is called a REDO record. We assume that log records are not coalesced across transactions; i.e., every (committed) page update has a corresponding REDO record. Log records are ordered and identified by a monotonically-increasing log sequence number (LSN). Transaction commit is recorded in the log using a commit record. The transaction commit record LSN determines the transaction serialization order.

Periodically, the database performs *checkpoints*. A checkpoint C writes page versions committed prior to some chosen checkpoint LSN (LSN_C) to the database

disk, and then records the checkpoint LSN in the log. To simplify the description, we assume the database only performs writes during a checkpoint. Retro, however, supports general database write policies (i.e. writes due to cache pressure) not described here, for lack of space.

We assume the database follows a no-STEAL writing policy [6]. This means that the database never writes uncommitted updates to disk. The recovery, therefore, only needs to REDO updates that were committed but not yet written since the last checkpoint; a database with a STEAL writing policy must also *undo* changes to pages which were not yet committed before the crash. Large memories render STEAL policy less important.

Checkpoints bound recovery time; the database does not need to recover updates that have been written to the database disk. The database is free to garbage collect the log entries prior to LSN_C, and begins recovery from LSN_C (to be more precise, the database starts recovery at, and can garbage collect log entries older than, the LSN of the oldest transaction that began before LSN_C and committed after LSN_C).

After a crash, BDB enters *recovery* upon being restarted. The database cannot re-enter normal operation until recovery has completed. During recovery, the database *replays* committed updates by applying REDO records from the WAL in LSN order since the last checkpoint. A successful recovery ends with a checkpoint; after recovery, the on-disk state of the database reflects the history of transactions that committed prior to the crash.

6.2 Snapshot recovery details

Retro expects to be invoked when BDB recovery applies REDO records. To identify updates tagged in OWS Retro must order updates relative to snapshot declarations. Retro records snapshot declarations in the log during normal execution, using a special log record (a *snaprec*). Retro expects to be invoked when recovery encounters snaprecs so that Retro can handle them (i.e. re-declaring the snapshot if needed).

Because BDB uses page-level concurrency control (i.e., two overlapping transactions may not both commit an update to the same page), we know that REDO records for a page P will be applied in transaction commit order. Retro invocations from update replay and snapshot declarations therefore run in transaction commit order, making it easy to identify updates tagged in OWS.

Algorithm 1 shows how the write-ordering invariants are enforced during a checkpoint. Whenever the database initiates a checkpoint, Retro takes control and finishes writing any snapshot data that had not yet been trickled to disk, and then updates snapshot metadata atomically using a database transaction. Finally, Retro returns control to the database, allowing the checkpoint to proceed normally.

Algorithm 1 Retro extension to database checkpoint

1: Pause database checkpoint
2: Write unwritten snapshot data to disk
3: Transactionally update snapshot metadata created since the last database checkpoint
4: Allow database checkpoint to proceed normally

Retro allows snapshot data to be trickled during normal operating periods (snapshot data is large and grows in proportion to the number of updates, so it is impractical to buffer all snapshot data created between checkpoints). After a crash, some pre-states may be on the Retro disk that have no corresponding snapshot metadata but the reverse can never be true due to snapshot-data-before-metadata write invariant. This means that when Retro recovery begins, any snapshot data written since the last checkpoint can be deleted by deleting any pre-states that are not referenced from snapshot metadata.

Algorithm 2 shows the two stages of Retro recovery. Stage 1 of Retro recovery resets snapshot data and metadata to a consistent state; then, Stage 2 runs alongside BDB recovery, saving a needed pre-state (if it has not been saved already). Database recovery ends with a checkpoint, the completion of which marks the completion of a successful recovery. Retro enforces the write invariants WAS and snapshot-metadata-before-data during this checkpoint, just like during normal operation. So, the checkpoint that terminates a successful database recovery for history H also marks the end of Retro recovery, at which point the on-disk state of Retro will reflect OWS(H).

Retro needs to *suppress* duplicate re-creation of snapshots in repeated recovery. Retro will know to correctly suppress re-creation of snapshots because after Stage 1, snapshot metadata and data will consistently reflect the latest snapshot that Retro has declared and the last pre-state it has saved before the crash. In Stage 2 therefore, Line 10 will suppress a duplicate snapshot declaration (A later snapshot is already present in Start). Line 17 will suppress duplicate saving of a pre-state (SavedAfter indicates the pre-state has been already saved).

We have considered a simplified writing policy that only writes database pages and Retro metadata at checkpoint time. Retro also supports more flexible writing policies. In particular, if the database is forced to write database pages between checkpoints due to cache pressure, Retro can still enforce its writing invariants by writing the pre-states of the pages forced from the cache (and making durable the associated snapshot metadata), and recover correctly if there is a crash before the next checkpoint. Suppression will still work correctly in this case because it is applied per-page based on a lookup in

Algorithm 2 Retro recovery

1: Recover snapshot metadata ▷ begin Stage 1
2: Delete unreferenced pre-states from Pagelog
3: S_{latest} := latest snapshot id in Start ▷ begin Stage 2
4: Normal database recovery begins replay
5: **repeat**
6: **if** Recovery requests page P for updating from the cache **then**
7: Make a copy of the pre-state of P in the cache
8: **end if**
9: **if** Recovery encounters a snaprec for snapshot S **then**
10: **if** $S > S_{latest}$ **then**
11: Declare S
12: $S_{latest} := S$
13: **end if**
14: **end if**
15: **if** Recovery encounters a commit record for transaction T **then**
16: **for all** Pre-states P created by T **do**
17: **if** $SavedAfter(P) < S_{latest}$ **then**
18: Save P for S_{latest}
19: **end if**
20: **end for**
21: **end if**
22: **until** REDO recovery is complete
23: Checkpoint database ▷ Retro will enforce the WAS invariant

SavedAfter, just like during normal operation.

7 Implementation Issues

Retro protocol extensions are implemented as a set of callbacks invoked from BDB transaction commit, recovery, and buffer cache protocols. There are two concerns in the way of an efficient implementation of the protocol extensions. Extensions that run concurrently, accessing shared mutable state, must be thread safe, and must be serialized in transaction order to correctly save and access snapshots. However, the implementation needs to avoid blocking transactions to synchronize extensions. Second, although storing Retro metadata in transactional structures simplifies snapshot recovery, frequent updates to metadata are costly. So, the implementation needs to reduce the cost of metadata updates.

7.1 Latest Snapshot

Extensions that declare snapshots, and save pre-states and metadata, invoked at commit, need to read and update the latest snapshot id. These extensions need to be serialized in commit order to correctly assign snapshot ids (numbered sequentially in declaration commit order), and to correctly identify updates tagged in OWS (is this the first update to P following the latest snapshot declaration?).

Extensions that create persistent data (e.g. save pre-states) must run after the invoking transaction commits (i.e. records its commit record in the log). The problem arises because concurrent transaction threads that block to write their commit records, get unblocked by BDB in arbitrary order (possibly after BDB runs a group commit), thus reordering the execution of extensions.

An extension must therefore determine the latest snapshot declaration preceding its transaction regardless of extension execution order. Instead of tracking the latest snapshot in a shared counter, Retro solves the problem by tracking recent snapshot declarations in a data structure called the SnapshotList. Entries for transactions that have completed the log write and therefore were assigned a commit LSN are sorted by their commit LSN. Retro uses the SnapshotList to assign snapshot ids sequentially in transaction commit LSN order regardless of extension execution order. An extension can determine the last-declared snapshot S_{last} for its invoking transaction T using the SnapshotList. It simply searches for the snapshot declaration with the highest LSN preceding the commit LSN of T.

7.2 SavedAfter cache

After saving a pre-state, Retro needs to update SavedAfter, to note it has been saved. Updating SavedAfter is expensive because it is a transactional data structure, so we describe a specialized write cache called the SavedAfter cache (SAC) that allows deferring updates to SavedAfter and consistent checking of SavedAfer.

In the Retro recovery protocol, Retro metadata is recovered first, before the application database, which means that the log into which SavedAfter updates are recorded must be separate from the BDB transaction log. As a consequence, when the commit extension needs to update SavedAfter(P), updating the durable SavedAfter metadata structure would require a second commit and log write, in addition to the commit of the application's transaction. We have measured the impact of committing an update to SavedAfter after saving a pre-state in commit extension and unsurprisingly, it has a significant impact on transaction throughput and latency.

To reduce the impact, we introduce an in-memory cache, called the SavedAfter cache (SAC), to store updates to SavedAfter on cache pages. Updates are propagated from SAC to SavedAfter when Retro data and metadata are flushed to the Retro disk. SAC therefore eliminates the SaveAfter update cost from the transac-

tion commit path, allowing multiple SavedAfter updates to be combined into a into a single transaction, and multiple updates to the same entry (e.g., SavedAfter?) to be absorbed.

Because the entries in SAC may be newer than those in SavedAfter, Retro must check SAC when looking up SavedAfter(P). Just as with SavedAfter, the SAC is a frequently-accessed data structure. However, SAC is not a source of extra contention in the system, because it leverages MVCC page-level locking and low-level concurrency mechanisms in the page cache.

SAC is implemented by extending the BDB page cache structures. A page P cached in the BDB page cache is preceded by a page header that contains meta information about the page such as its name, the commit LSN of the last transaction to update the page, the linked list of pages that form the version chain (VC) for this page, maintained by MVCC, and other internal metadata. SAC(*P*) is stored on the header for cached page P (requiring that every page header be enlarged by the size of a snapshot id).

SAC(*P*) is initialized from SavedAfterCache(P) when there is a cache miss and P is read from disk. When MVCC creates a new version of page P in the cache for an update to P, SAC of the new version is initialized by the commit extension of the transaction that commits the update. When checking whether to save the pre-state, the commit extension uses the SAC value on the pre-state. If the pre-state of P needs to be saved for snapshot S, the SAC on the new version of P will be set to S. Otherwise, the SAC on the new version of P will be copied from the pre-state of P.

7.3 SAC and Retrospection

Retro runs a retrospective query as MVCC transaction T. An extension invoked from T observes the Vsnap of T. Consider a retrospective query T as off snapshot S, serialized after transaction execution H. A page translation extension, invoked when T accesses a page P, will observe a consistent SAC(*P*) value as of the begin LSN of T, reflecting all tagged update commits and effects of their associated extensions that precede T in OWS(H). If the SAC value indicates P@S is shared with the database, (i.e. it was shared when T began) the translation will correctly direct the access to the version of P from *Vsnap* of T. Otherwise the translation will redirect the access to *P@S* saved by Retro, as required by OWS specification.

8 Performance Evaluation

Our simple study evaluates the performance of retrospection, the new feature, and Retro overheads to BDB transactions. The results confirm the low overhead of Retro to BDB. The results also show a slowdown for running Retro transactions (i.e., transactions that use snapshots); reducing these overheads is an area for future research.

The Retro prototype extends BDB version 5.3.21. The prototype has just over 5000 lines of C code. The modifications to BDB to integrate Retro are minimal: under 250 LOC were modified or added to BDB source code. The implementation includes the complete design for the C API used in the evaluation. We are in the process of completing the SQLite API. Preliminary results using SQLite API confirm the results from the C API.

The hardware platform is a quad-core Intel Xeon CPU at 2.66ghz with 4 gb of physical RAM and 2 Seagate Cheetah 15,500 RPM SAS hard drives, running DEBIAN GNU/Linux version 2.6.32. BDB stores database files in the file system, formatted with ext3. BDB uses default page size of 4K. Disk level prefetching is disabled, to emphasize the cost of random disk I/O for Pagelog. Our platform only supports two disks, so we use one for the database, Pagelog, and metadata, and one for the (separate) database and metadata logs (BDB database and log must be on separate disks since otherwise BDB (without Retro) transaction throughput drops to zero during a checkpoint). This is sub-optimal configuration for Retro, since Pagelog could impact BDB reads and writes. Our experiments use in-memory working sets for the database, insulating database reads from Pagelog I/O. Moreover, since Retro trickles snapshots to disk between checkpoints (unless noted otherwise), this is not a problem for database writes in our experiments.

All measurements are reported as the average of 10 runs (non-negligible standard deviations are depicted).

8.1 Retrospection

Page name translation. We evaluate the performance of Retro transactions using a simple read query *Q* that fetches 2048 random records (each 108 bytes in size) from a table cached in memory. We report *slow-down*, the measured time-to-completion relative to running *Q* in-memory in unmodified BDB. The workload maximizes the CPU overhead of Retro transactions since *Q* performs no computation.

The CPU overhead of page name translation comes from looking up the page in SavedAfter (the current state page is not cached, no SAC) or SAC and, if the snapshot page has been already saved as a pre-state, looking up the location of that pre-state in SPT.

Figure 5 shows that a slow-down when accessing pages shared with the current state is 1.6x (“Retro(Q) cur”) using SAC. This overhead is similar in magnitude and origin to the one that was found due to the buffer pool indirection and latching for tree lookups in Shore [2]. The slow-down when accessing snapshot pages saved in

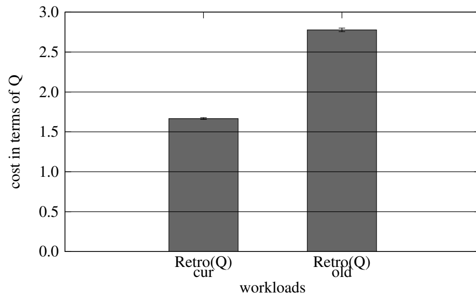

Figure 5: CPU costs

Component	*Cost*
SPT(S) get	0.55*us*
SavedAfter get	47*us*
SAC get	0.06 *us*
SPT(S) put	11*us*
Pagelog get	3.62 *ms*

Figure 6: Translation costs

Pagelog ("Retro(Q) old") is about a 2.7x, due to additional access to SPT, including lookup and insertion of mappings into the SPT by scanning Maplog. The slowdown when accessing Pagelog is higher when the pages accessed by retrospection do not have a current state version cached in memory (no SAC), and therefore a costlier lookup in SavedAfter is needed. The dominant cost however is accessing Pagelog.

The absolute costs of translation components are shown in Table 6. Insert into the SPT is costly, more so than lookup. The SPT is implemented using a simple hash table that is not optimized for resizing, and resizing is frequent during Maplog scan. Maplog in-memory scan is costly. Maplog is composed of many small mappings that must be individually searched. Maplog is not shown in the table because it does not have a clear per request cost. The total contribution of costs from Maplog resembles total contribution of SPT insert. Optimizing access to SPT and Pagelog is an area of future work.

Snapshot page I/O. I/O from the Pagelog has different performance characteristics than I/O from the current state database. Copy-on-write declusters snapshot pages, resulting in a different spatial layout from the current state. A sequential query incurring sequential disk I/O costs in the current state can perform poorly when running retrospectively, incurring random disk I/O costs, similar to I/O in a log structured file system [11]). Large memories and SSDs can eliminate the declustering penalty, and we plan to use SSD for Retro in future work. Nevertheless, for large volumes of snapshots, the SSD solution may not be practical. Since declustering effect for sequential queries is well understood, here we

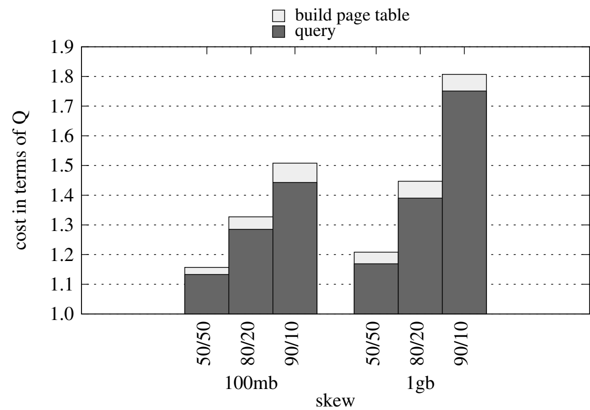

Figure 7: I/O-bound retro query costs

focus on random queries to expose an additional source of Pagelog performance difference not directly related to declustering. When the update workload is non-uniform (skewed), pages from the same snapshot may be very far apart. By modeling the skew in the update workload, we can characterize the I/O overhead of random queries run retrospectively. We use a standard model of skewed workload: "80/20" means 80% of the updates go to 20% of the database, "50/50" means the workloads is not skewed. The portion of Pagelog corresponding to any snapshot fits on a single disk; the seek impact may actually be mitigated if Pagelog is spread over multiple disks.

Figure 7 compares the cost of I/O for $Retro(Q)$ for different update skews to the cost of I/O for Q in BDB. The experiment runs with cold caches. The query for this experiment is identical to the "Retro(Q) old" query depicted in figure 5, except that the cache starts cold (we report "slow-down" due to Retro as a ratio; the absolute latencies for I/O-bound queries were 3 to 4 orders higher than in the CPU-bound experiments). We include breakdown for snapshot data and metadata. The full analysis of Skippy index appears in [14].

The skew impact persists independently of table size. We run the experiment with 100MB table and a 1GB table (figure 7). We scaled up the number of random records read in the query for the larger table so that in both cases the query reads 1% of the table. For both the small and large tables, there is a similar trend of increasing cost of $Retro(Q)$ relative to Q as skew increases. The impact of skew appears higher in the larger database, but this is due to the decreased hit ratio in the larger query. In the large table Q had a 7% hit ratio in the leaf pages of the table, as opposed to 14% in the small table (the cache starts cold, a single page read pre-fetches multiple records, and given a constant page size, the likelihood of a subsequent hit on the pre-fetched page decreases as the

table size increases because a smaller fraction of table records are clustered on each page).

8.2 Overhead to BDB

Retro adds no direct overhead to BDB queries. To updates, Retro adds commit time CPU overhead (SAC must be checked and possibly updated for each update), and possibly synchronous I/O (to log a snapshot declaration record). Enforcing the Before-GC invariant (section 5) requires writing pre-states in the background but does not delay commits. Enforcing the WAS invariant (Sec 6) can increase checkpoint latency due to snapshot I/O. Trickling snapshots between checkpoints avoids the checkpoint delay. Writing snapshots to a different disk avoids contention between Retro and the current state during checkpoints altogether. Our system has two disks: one is used for the transaction log, so Pagelog and current state share a disk.

We run a simple micro benchmark to test the overhead to update transactions running in memory incurred by saving snapshot pre-states in-memory and writing all snapshot data and metadata to disk during checkpoints. We ran the random update transaction described in section 8.1 in Retro, and unmodified BDB. With Retro, the system declares a snapshot after each update transaction, saving pre-states for the pages modified by 1000 random updates on each transaction commit, thus maximizing the number of snapshot pages that must be saved. We calculate throughput from the time-to-completion and the number of completed update transactions. We observe an overhead of about 4% to update throughput in our workload from Retro, confirming previous results [15] concerning the low impact of the split snapshot writes.

9 Conclusion

We described Retro, a new efficient system for retrospection in Berkeley DB, implemented using a new simple and robust method that avoids invasive database modifications. Our approach is adapted to BDB. However, because WAL, MVCC and buffer cache are standard protocols, we believe the approach is more general and we are investigating other extensions in on-going work. The key challenges going forward are optimizing the performance of retrospective queries and supporting SSDs.

References

[1] BRIDGE, W., JOSHI, A., KEIHL, M., LAHIRI, T., LOAIZA, J., AND MACNAUGHTON, N. The Oracle universal server buffer. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1997), Morgan Kaufmann Publishers Inc., pp. 590–594.

[2] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. OLTP through the looking glass, and what we found there. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 981–992.

[3] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. In *Foundations and Trends in Databases* (2007), vol. 1.

[4] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 19–19.

[5] LOMET, D., BARGA, R., MOKBEL, M. F., SHEGALOV, G., WANG, R., AND ZHU, Y. Immortal DB: transaction time support for SQL server. In *Proceedings of the ACM SIGMOD international conference on Management of data* (2005), pp. 939–941.

[6] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17* (March 1992), 94–162.

[7] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A. P., AND ZADOK, E. A versatile and user-oriented versioning file system. In *FAST* (2004).

[8] ORACLE CORPORATION. Berkeley DB degrees of isolation. `http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/transapp_read.html`.

[9] OZSOYOGLU, G., AND SNODGRASS, R. T. Temporal and real-time databases: A survey. *Knowledge and Data Engineering 7*, 4 (1995).

[10] PLATTNER, C., WAPF, A., AND ALONSO, G. Searching in time. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 754–756.

[11] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[12] SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A., CARTON, R., AND OTIR, J. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles* (1999).

[13] SHAULL, R. *Retro: A methodology for Retrospection Everywhere*. PhD thesis, Brandeis University, Aug. 2013.

[14] SHAULL, R., SHRIRA, L., AND XU, H. Skippy: a new snapshot indexing method for time travel in the storage manager. In *SIGMOD Conference* (2008).

[15] SHRIRA, L., AND XU, H. Snap: Efficient snapshots for back-in-time execution. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society.

[16] SHRIRA, L., AND XU, H. Thresher: An efficient storage manager for copy-on-write snapshots. In *USENIX '06: Proceedings* (Berkeley, CA, USA, 2006), Advanced Computer Systems Association.

[17] WIRES, J., AND FEELEY, M. J. Secure file system versioning at the block level. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), ACM, pp. 203–215.