Skippy: a New Snapshot Indexing Method for Time Travel in the Storage Manager

Ross Shaull, Liuba Shrira, and Hao Xu
Department of Computer Science, Brandeis University
Waltham, Massachusetts, USA
rshaull@cs.brandeis.edu, liuba@cs.brandeis.edu, hxu@cs.brandeis.edu

ABSTRACT

The storage manager of a general-purpose database system can retain consistent disk page level snapshots and run application programs "back-in-time" against long-lived past states, virtualized to look like the current state. This opens the possibility that functions, such as on-line trend analysis and audit, formerly available in specialized temporal databases, can become available to general applications in general-purpose databases.

Up to now, in-place updating database systems had no satisfactory way to run programs on-line over long-lived, disk page level, copy-on-write snapshots, because there was no efficient indexing method for such snapshots. We describe Skippy, a new indexing approach that solves this problem. Using Skippy, database application code can run against an arbitrarily old snapshot, and iterate over snapshot ranges, as efficiently it can access recent snapshots, for all update workloads. Performance evaluation of Skippy, based on theoretical analysis and experimental measurements, indicates that the new approach provides efficient access to snapshots at low cost.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.2.2 [**Database Management**]: Physical Design

General Terms

Design, Performance

1. INTRODUCTION

The storage manager of a general-purpose database system can retain consistent disk page level snapshots and run application programs "back-in-time" over long-lived past states, virtualized to look like the current state. This opens the possibility that functions, such as on-line trend analysis and audit, formerly available in specialized temporal databases, can become available to general applications in general-purpose databases.

Up to recently, a general purpose database system that updates data in-place, had no efficient way to support on-line back-in-time execution over snapshots. Retaining frequent long-lived snapshots was simply too disruptive to the database performance [17], and back-in-time execution (BITE) over long-lived copy-on-write snapshots has been prohibitively slow. Split snapshots [21, 22, 20] is a recent approach that overcomes some of the limitations of earlier systems and sets the context for our work. The approach provides snapshots in a form that virtualizes the database storage, allowing a database to run unmodified application programs and access methods over consistent snapshots on-line (i.e., in production, not in the warehouse). The snapshot system is integrated at the buffer manager level so that consistent copy-on-write snapshot pages can be retained without the need to quiesce the database, avoiding disruption. The snapshot pages are written in a separate snapshot store, supporting update-in-place.

An open problem has been how to provide efficient on-line program code access to long-lived copy-on-write snapshots when snapshots are retained for a long time. The problem is that the code needs an efficient way to find the snapshot pages that belong to a given snapshot. Some snapshot systems use the recovery log and "roll back" to a consistent snapshot. Such solution is acceptable in a short-lived snapshot system but would be inefficient in a long-lived system. In contrast, most existing access techniques to versioned data in databases [11] and file systems [23, 18] rely on no-overwrite update. The past state remains in-place and the new state is copied, so the access structure for the past state just takes over the access structure for the current state.

The split snapshot system [21] indexes the copy-on-write snapshot pages at low-cost by writing the mappings of the snapshot pages into a sequential log as it copies snapshot pages to the snapshot store. It scans the mapping log when a given snapshot page is needed. Scanning page mappings is faster than scanning a recovery log. A mapping scan can be still slow if some database pages are modified infrequently, since the scan has to pass over many repeated mappings of the frequently modified pages before finding mappings for infrequent ones. Yet, both infrequently and frequently modified data is common. The code requesting to run on a snap-

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

shot has to wait for the scan to encounter mappings for both frequently and infrequently modified pages. Caching infrequent mappings in-memory can accelerate the scan [21], but that approach does not scale as snapshot lifetimes increase.

Our system solves this problem. We describe a new efficient indexing method called Skippy that hierarchically builds condensed persistent summaries of the mapping log, with duplicate entries removed, and links the summaries into the mapping log at regular intervals. Slow scans can now proceed faster over these lower-resolution summaries. An application may need to access multiple snapshots, for example to analyze a trend. In addition to back-in-time execution of programs "within" a snapshot (BITE), Skippy supports efficient "across time" execution (ATE) of programs written in the Map-Reduce style [4]. Given a range of snapshots, an ATE program runs BITE over each snapshot, applying the Map function, and composes the results, applying the Reduce function. Using Skippy, a database application can run on-line over arbitrarily old snapshots, and iterate over snapshot windows, as efficiently as it can access recent snapshots, for any database update workload.

Running BITE on a snapshot resembles an "as-of" query in a transaction time database [16], running ATE over a range of snapshots resembles a computation of a SQL aggregate over a past data stream [7, 3]. By accelerating BITE and ATE over snapshots, Skippy serves a similar purpose to a multiversion access method that indexes logical records, albeit at a different level in the DBMS software stack, and using a radically different method based on scanning mappings to construct a page table, instead of searching an ordered set at each access. Nevertheless, like the state-of-the-art methods for as-of queries [16], Skippy guarantees that access to a snapshot remains efficient independent of snapshot age and update workload.

The formal model underlying Skippy relates snapshot consistency to the order in which snapshot page mappings are written, decoupled from the order in which snapshot pages are written. Relating snapshot consistency to the order of mappings provides an important practical benefit. Efficient split snapshot implementations in third party database software become possible, since the small mappings can be ordered in-memory by tracking the database buffer manager write policy, instead of dictating it [21]. This extends the benefits of BITE and ATE beyond experimental systems [21, 22, 20], and we describe a Skippy prototype in the commercial strength BDB in our experimental evaluation.

We demonstrate Skippy effectiveness both analytically and experimentally (using implementations in two different split snapshot systems). Theoretical analysis using a standard hot/cold workload model shows that Skippy can counteract the impact of infrequently-modified pages on scan time while imposing modest requirements on the database. Our split snapshot system integrated into Berkeley DB, SkippyBDB, demonstrates that this can be achieved using a similar workload model in a real system (section 5.3). Our SkippyBDB experiments also agree with our analysis by showing that creating Skippy imposes low overhead on the database. Our implementation of Skippy in the SNAP [21] split snapshot system, using an application-level OO7-based benchmark [2], further demonstrates that Skippy performs well, providing up to a 19-fold performance improvement (section 5.2).

The problem of efficient access to long-lived past states is an old problem. Skippy-based split snapshots offer a novel solution that potentially could be adapted to become a standard DBMS feature. Our BDB prototype shows a step in this direction. In a full-strength DBMS system, such an approach would require more effort but is attractive because, as a system architecture, it is considerably less complex and invasive, and more general, than the widely-known alternative of capturing past states and providing multiversion access methods at the logical database level [24, 8].

This paper makes the following contributions: (i) presents a new efficient indexing method for long-lived split copy-on-write snapshots, solving an open problem, and providing a first practical solution for on-line code access to long-lived past states in a general database, for general computations, running both within and across snapshots, (ii) describes a consistent snapshot model underlying the method, (iii) analyzes theoretical performance and suggests how to achieve efficient indexing in practice, (iv) provides a prototype implementation and experimental measurements supporting the claims in the analysis.

2. LONG-LIVED SPLIT SNAPSHOTS

In a split snapshot system [21, 22, 20], an application takes a snapshot by issuing a snapshot *declaration request*. The system serializes the snapshot in transaction order, returning to the application a *snapshot name*. For simplicity, we assume snapshots are named in an increasing integer sequence order. Snapshots are *consistent*, i.e., a snapshot v reflects all the modifications committed by the transactions that precede the declaration of v, and none of the modifications committed by the transactions that follow the declaration of v.

Consider a database storage system that includes a set of disk pages P1, P2, ..., Pk, and a page table that maps the database pages into their disk locations. The snapshot system virtualizes database storage, adding a layer of indirection between the physical address of disk pages and the database paging architecture, similar to shadow tables [5]. A snapshot consists of a set of *snapshot pages* and a *snapshot page table* that maps snapshot pages to their disk locations. Virtualizing database storage enables efficient back-in-time execution (BITE), a system capability where programs running application code and all access methods can run against consistent snapshots, transparently accessing snapshot pages the same way as database pages.

The snapshots are stored in a log-structured storage system component called *snapStore*, on a separate disk, for best performance. A snapshot is copied into the *snapStore*, page-by-page. A specialized copy-on-write mechanism captures in-memory enough pre-state of a page P just before an update overwrites P for the first time after the declaration of snapshot v. The captured update pre-states are used to generate consistent snapshot pages when the buffer manager writes the corresponding updates to the database disk. The snapshot pages are written in parallel to the *snapStore* disk. The split snapshot system leverages the existing database recovery mechanism to safely write cached pre-states in the background [20]. Performance evaluation [21, 22] shows that this snapshot approach has low performance impact.

Figure 1 shows a database containing 3 pages ($P1$, $P2$, and $P3$) the database page table (PT), and the snapshot page

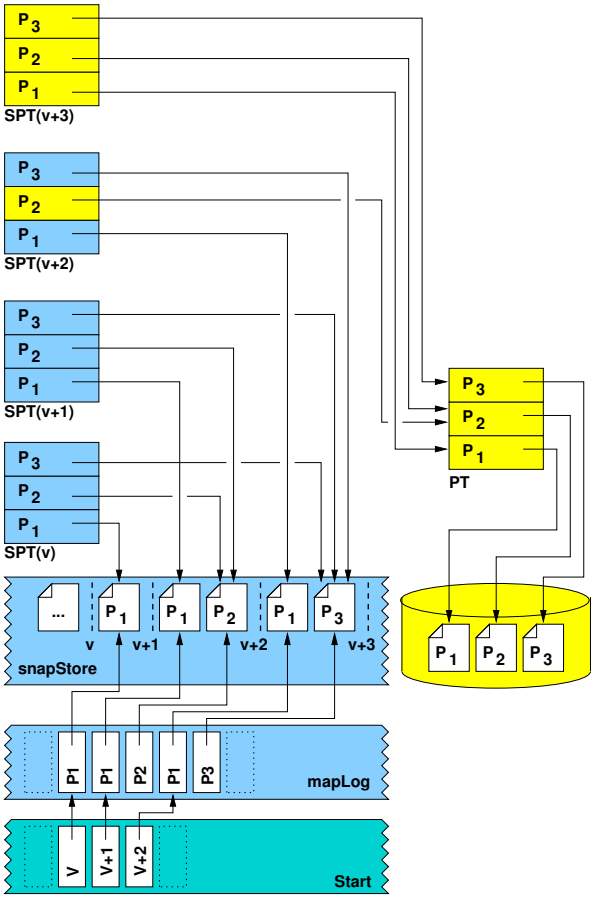

Figure 1: Database storage virtualized for BITE

tables (SPTs) for 4 snapshots ($SPT(v)$–$SPT(v+3)$). The following history of 4 transactions (T1, T2, T3, and T4) is depicted. T1 declares snapshot v and modifies $P1$, resulting in a pre-state for $P1$ being retained for v in *snapStore*. T2 declares snapshot $v+1$ and modifies $P1$ and $P2$, resulting in pre-states for $P1$ and $P2$ being retained for $v+1$ in the *snapStore*. T3 declares snapshot $v+2$ and modifies $P1$ and $P3$. T4 declares snapshot $v+3$ but does not modify any pages.

The pages in *snapStore* are pre-states captured using copy-on-write. Because pre-states are captured incrementally not all snapshot page tables have all of their entries pointing into *snapStore*. For example, no database pages have been modified since snapshot $v+3$ was declared, so all of the entries for $SPT(v+3)$ point into the database (the *current state*). Likewise, only $P1$ and $P3$ have been modified since the declaration of snapshot $v+2$, so while the entry for $P1$ and $P3$ point to pre-states in *snapStore*, the entry for $P2$ in $SPT(v+2)$ points into the database. If a subsequent transaction modifies $P2$, then both $SPT(v+2)$ and $SPT(v+3)$ will have their entries for $P2$ updated to point into the pre-state for $P2$ copied into *snapStore*. After this, snapshots $v+2$ and $v+3$ will *share* the same pre-state for $P2$, just as v and $v+1$ share the pre-state for $P2$ retained for $v+1$, as depicted in figure 1.

SPTs change over time; and, there may be many of them (in order to support high-frequency snapshots). It could be expensive to maintain a sorted snapshot page table on disk for every snapshot in the system. Instead, we construct SPTs for a snapshot when BITE on that snapshot is requested.

The *snapStore* resembles a page level pre-state log. Consider such a log with page states ordered in increasing time (transaction) order. A snapshot is comprised of pages appearing in this log after the point at which the snapshot was declared. An inefficient way to build a snapshot would be to apply the pre-state log backwards until reaching the snapshot point. A better way is to scan the pre-state log forward from the snapshot point, stopping at the first entry for each needed page. If no entry is found, the page has not changed since the snapshot declaration, and the current page version can be used. The drawback of this approach for running code is that finding the needed page content can still require scanning arbitrary large amounts of the pre-state log.

The SNAP split snapshot system [21] addresses this drawback by replacing the scan of the snapshot page log with a scan of a log of pointers to the snapshot pages. Scanning the log of pointers (page mappings) requires substantially fewer disk reads. Furthermore, the cost of creating the mapping log, called *mapLog*, is low. Figure 1 depicts the *mapLog* created during the execution of transactions T1, T2, T3, and T4, as well as *Start*, which points to the first mapping written into *mapLog* for each of the snapshots v to $v+2$ ($v+3$ has no mappings in *mapLog* yet, so no *Start* pointer for $v+3$ is depicted). Since snapshots are indexed by sequence number, Start can be accessed like an array in constant time.

The problem is that when some pages are modified much more frequently than others, if an unmodified page P lives a long time after a snapshot was declared before it is overwritten, a scan to find P has to pass over large number of duplicate mappings that correspond to frequently modified pages. Yet, update workloads containing both frequently and infrequently modified pages are common. For example, consider a program that manages a large collection of items, and allocates on page P a variable to hold the size of the collection. Some items and pages may be modified infrequently but if the size of the collection changes frequently then the workload will be skewed, and with frequent snapshots mappings for page P will appear many times in *mapLog*, lengthening any scan over *mapLog*. Access to infrequently modified pages introduces significant delays in the code running over snapshots on-line and moreover, such accesses may be frequent. The SNAP system addresses this problem by keeping the mappings of infrequently modified pages in-memory. The solution however does not scale as snapshot lifetimes increase.

3. MAPPING CONSISTENCY

Mapper is the split snapshot system component that tracks the location of the snapshot pages. Mapper method *write* inserts snapshot page mappings into the sequential persistent data-structure *mapLog*. Mapper method *lookup* searches the *mapLog* for the mapping of the requested snapshot page.

Consider a pre-state of a page, corresponding to the first modification to a page committed after the declaration of snapshot v, and before the declaration of snapshot $v+1$. This pre-state belongs to snapshot v. Call such a pre-state a *page*

retained for snapshot v. Without constraining the snapshot page copying order, the Mapper write method enforces the following invariant:

$\mathbf{I_{mapLog}}$ Invariant *all the mappings for pages retained for snapshot v are written into mapLog before all the mappings for pages retained for snapshot* $v+1$.

Let $start(v)$ be the first mapping in the *mapLog* for a page retained for a snapshot v, and let FEM be a shorthand for *first encountered mapping*. The following theorem underlies the correctness of the Mapper lookup method.

FEM Theorem *the FEM for page P in a sequential scan of mapLog starting from* $start(v)$ *onward corresponds to a version of page P that belongs to snapshot v.*

The proof of the FEM theorem directly follows from the invariant I_{mapLog} that insures the FEM for P corresponds to the page pre-state that was captured when the page P was modified for the first time after the snapshot v declaration, and is, therefore, the correct page P version corresponding to snapshot v.

The Mapper lookup method therefore finds a mapping for a page P in snapshot v by sequentially scanning the *mapLog* from the position $start(v)$ onward, returning the first mapping it encounters for a page P.

Consider the transaction history depicted in figure 1. In order to construct the snapshot page table for snapshot v ($SPT(v)$), Mapper lookup begins with the first mapping retained for v (pointed to by $Start(v)$), and scans forward, copying FEMs into $SPT(v)$ as they are encountered. The FEMs for v in this example are the mapping to the pre-state of $P1$ retained for v, the mapping to prestate $P2$ retained for $v+1$, and the mapping to prestate $P3$ retained for $v+3$. The second two mappings to $P1$ in *mapLog* (to the pre-states of $P1$ retained for $v+1$ and $v+2$) not FEMs for $SPT(v)$, so they are ignored during the scan.

The Mapper lookup is guaranteed to find all the mappings for a given snapshot provided the entire snapshot state has been copied into the snapshot store. The condition holds if the entire database state has been overwritten since the snapshot declaration. In a long-lived snapshot system, this condition will hold for all the snapshots older than some given threshold that advances with time. Our discussion below considers only such older snapshots. We consider recent snapshots in Section 4.4.

4. SKIPPY MAPPER

Because mappings are small compared to snapshot pages, and since they can be written with background i/o, writing *mapLog* sequentially to disk has low impact on snapshot performance. In contrast, the lookup function incurs a foreground cost since the application requesting the snapshot must wait for the lookup to complete. To support efficient BITE that runs application code on a snapshot by transparently paging in snapshot pages, the entire snapshot page table for a snapshot is constructed when application requests a snapshot. To do this, the system needs to find in the *mapLog* all the mappings for a given snapshot.

Some BITE applications may only need to access a small subset of the pages in a snapshot, so constructing the complete snapshot page table could be wasteful. Our approach supports an "on-demand" version of the Mapper lookup that avoids "in-advance" complete construction instead scanning incrementally and Section 5.3 evaluates the benefits of an on-demand Mapper. We also support a "cold" Mapper that avoids constructing a page table, scanning anew on each lookup. However, since the order of the page mappings in the *mapLog* is determined by the page overwriting order, in the worst case, the lookup of any single page may require as much effort as the construction of the entire snapshot page table. Moreover, the approach we will describe benefits the on-demand and cold lookup as well. Therefore, without loss of generality, unless specified otherwise, the discussion below assumes that the Mapper lookup constructs the entire snapshot page table.

The length of the scan collecting the FEMs for a snapshot v is determined by the length of the *overwrite cycle* of a snapshot v, defined as the transaction history interval starting with the declaration of snapshot v, and ending with the transaction that modifies the last database page that has not been modified since the declaration of snapshot v. Once the overwrite cycle for snapshot v is complete, all database pages corresponding to the snapshot v will be copied into the snapshot store and therefore the mappings for all the pages will be entered into *mapLog*.

When the update workload is uniform, most of the mappings read by the lookup scan for snapshot v are likely to be the first encountered mappings (FEMs). As explained earlier, in many storage systems the update workload is *skewed* so that some pages are updated significantly more frequently than others, resulting in an increase in the number of non-FEM mappings encountered during a scan of *mapLog*.

When the update workload is skewed, the lookup scan slows down. For example, even a mild skew, where a third of the database pages are modified by two-thirds of the page updates, doubles the length of the overwrite cycle as compared to a uniform workload, thus doubling the length of the construction scans (see Section 5.1). Since the application is waiting for the scan to complete, it is important to reduce the cost of the scan for skewed workloads.

4.1 Skippy structure

Skippy accelerates SPT construction for skewed workloads by allowing the construction scan to skip over the unneeded repeated mappings. Skippy collects the FEMs from *mapLog* into higher-level logs, omitting the mappings corresponding to frequently-modified pages. The logs are organized into nodes that form a forest of trees. The tree pointers, directed from the nodes in the lower-level logs to the nodes in the upper-level logs, guide the construction scan from the lower-level log containing many repeated mappings to the higher-level "fastlane" logs that contain fewer repeated mappings but still contain all the mappings needed for constructing SPTs.

4.2 2-level Skippy

Skippy is constructed on top of a *mapLog* that is subdivided into successive partitions, called nodes. The level-0 Skippy nodes n_0^0, n_1^0, ... are populated from the successive *mapLog* partitions. n_0^1, the parent node of n_0^0, is populated by copying into it the FEMs from n_0^0, followed by the FEMs from n_1^0, and so on, until the parent node is full. The copying then continues into the next level-1 parent node. Each level-0 node with FEMs copied up has a pointer called *uplink* that points to the next FEM written into its parent node (as

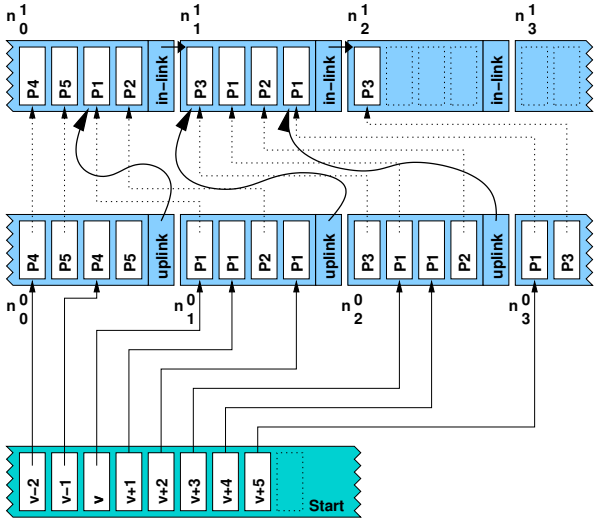

Figure 2: Example fragment of a 2-level Skippy

you will see, the scan which follows *uplinks* will encounter the same FEMs as a scan which ignores *uplinks* and scans only in *mapLog*). The roots of the tree, unlike the level 0 nodes, do not have an *uplink*. Instead, they are chained using a pointer called *in-link* that leads to the first mapping in the next root node (within the same level). This process constructs a two-level Skippy forest.

Figure 2 shows an example of a 2-level Skippy construction resulting from executing consecutive transaction history sequences H1, H2, H3 in a database with 5 pages. H1 declares snapshots $v-2$ and $v-1$, each snapshot has pages P4 and P5 retained for it. H2 is the transaction sequence depicted in figure 1, which declares snapshots v (retained P1), $v+1$ (retained P1 and P2), and $v+2$ (retained P1 and P3). H3 repeats H2, declaring respectively snapshots $v+3$, $v+4$, and $v+5$. The solid arrows in the figure are pointers to mappings within nodes. The dotted arrows indicate which mappings get copied into a higher level. The node n_0^1 contains the FEMs for P4 and P5 copied from n_0^0 (setting the *uplink* from n_0^0 to point right after the copied FEMs), and the FEMs for P1 and P2 copied from n_1^0 (setting the *uplink* from node n_1^0 to point to node n_1^1). Notice, in this skewed workload, the level-1 Skippy nodes contain half as many mappings for the frequently modified page P1, compared to the level-0 nodes.

The Skippy scan employs a simple map to locate the level 0 Skippy node n_i^0 containing $start(v)$, the first mapping recorded by snapshot v, and the location of this mapping within the node. Skippy scan reads in the node n_i^0 and collects the FEMs starting from the location identified by $start(v)$ to the end of the node. The scan then follows the uplink pointer in n_i^0, and proceeds at the parent node at level 1 to the end of the node and follows the in-link pointers through the successive root nodes.

For example, consider a Skippy scan, constructing $SPT(v-2)$ in the example 2-level Skippy shown in figure 2 and starting with an empty $SPT(v-2)$ (assuming completed overwrite cycle for $v-2$) at the node n_0^0 containing $start(v-2)$. The scan continues to the end of this node collecting FEMs for P4 and P5, follows the uplink pointer into the parent node n_0^1 collecting FEMs for P1 and P2, continues following the in-link pointer to the node n_1^1 collecting the FEM for P3, and scans following in-links until $SPT(v-2)$ is filled. Note, the construction of $SPT(v-2)$ avoids scanning three repeated mappings for the frequently modified P1 when using a Skippy scan. This benefit applies to any scan through node n_1^1 constructing a page table for a snapshot preceding $SPT(v-2)$. The following theorem states the correctness of the 2-level Skippy.

Skippy Theorem *the Skippy scan that starts at* $start(v)$ *constructs the correct snapshot page table for snapshot* v.

The proof of the Skippy theorem is by construction: the Skippy scan collects the same FEMs as would be collected by a scan that proceeds at the level-0 nodes without ever climbing to level-1. Since the level-0 scan collects the same FEMs as the basic *mapLog* scan, by the FEM theorem, the Skippy scan constructs correctly the snapshot v page table.

4.3 Multi-level Skippy

The 2-level Skippy scan accelerates snapshot page table construction compared to the basic *mapLog* scan because it skips repeated mappings that appear within a single level-0 node when it scans at level-1. Nevertheless, a scan that proceeds at level-1 will still encounter repeated mappings that appear in multiple level-0 nodes. In the example in Figure 2, the scan that constructs $SPT(v-2)$ and proceeds at level-1 encounters in node n_1^1 the repeated mappings for P1, copied from n_2^0 and n_3^0.

To eliminate repetitions over multiple level-0 nodes, the 2-level Skippy described above can be generalized, in a straightforward way, to a multi-level structure.

We construct a multi-level Skippy inductively. Given a $(h-1)$-level Skippy, we construct a h-level Skippy by treating the level $h-1$ nodes as level 0 nodes in the 2-level structure. That is, we construct the level h nodes the same way we have constructed level 1 nodes, by copying the FEMs from the level $h-1$ nodes. The copying eliminates repetitions among the mappings inside a level h node while retaining the FEMs. Like in the 2-level Skippy, all non-root nodes contain an uplink pointer pointing to the parent node, and the root nodes (level h) are chained via the in-link pointers.

The scan in the h-level Skippy starts at the level 0 node just like the 2-level scan, and proceeds to the end of the node before climbing to the next level. Similarly, after reaching a top-level node, the scan proceeds to follow in-link pointers through the successive root nodes.

The correctness argument for the Skippy scan in the h-level structure is inductive, following the construction and using the Skippy Theorem for 2-level structure as the base case. Namely, by construction, the FEMs collected by a scan after climbing to a level h following an uplink from a level $h-1$ node, are identical to the FEMs that would be collected if the scan continued at the level $h-1$. Since the scan at level $h-1$ collects the correct FEMs, so does the scan at level h.

Disk i/o costs. The Mapper writes the Skippy forest to disk by writing a separate sequential log for each level. The total Skippy Mapper write cost, therefore, includes the sequential disk i/o cost for the basic *mapLog* creation, and the additional cost to write mappings into all Skippy levels. Mapper write costs should be kept low to avoid impacting

the storage system. This issue is particularly important to support low-cost selective snapshot garbage collection [22] that replaces page copying with cheaper mapping copying, amplifying the cost of writing Skippy forest.

The Skippy scan performs sequential i/o while reading the mappings in a node and then following an in-link, but performs a disk seek when following an uplink. The cost of a Skippy scan depends on the number of levels, the size of the nodes, and the workload that produces the mappings. In Section 5 we analyze these costs.

4.4 Recent Snapshots

A recent snapshot with an incomplete overwrite cycle has some of its pages still residing in the database. If a page P for snapshot v resides in the database, a lookup scan will find no FEM for P in snapshot v simply because there is no mapping for P present in *mapLog* after $start(v)$.

Mapper avoids unnecessary searches using a simple data-structure called *lastRetainer*, which specifies if the page P is still in the database for snapshot v. *lastRetainer* keeps for each database page P the name of the most recent snapshot that has the page P retained for it, where we assume that snapshot names can be deterministically ordered (e.g., they are integers). If $lastRetainer(P) < v$, no search is needed because P for snapshot v is still in the database. If the pre-state for a page P is retained for snapshot v, then *lastRetainer* must be updated such that $lastRetainer(P) := v$, so that future lookups for P in v consult *mapLog*.

4.5 Across-Time Execution

In addition to running BITE against a single snapshot, an application may be interested in analyzing past states from a sequence of snapshots in a time range. Across-Time Execution (ATE) provides a convenient abstraction for efficiently executing code in a series of snapshots. ATE utilizes the *map* abstraction to execute code in each snapshot (using BITE), generating a set of results. ATE also allows programmers to provide a *reduce* callback which iteratively calculates a single result from the set of results returned by mapping BITE over each snapshot. *Map/reduce* is a common abstraction for list processing, and has also been applied successfully to processing large data sets in [4]. The framework for the *reduce* calculations is outside the scope of this work; we describe here how the *map* portion of ATE can be made more efficient by exploiting a unique property of Skippy.

Running BITE on consecutive snapshots using Mapper lookup can be wasteful, because the same mappings could be read multiple times. For example, consider the transaction histories depicted in figure 2. Running code over each of the snapshots declared by history H2 using Mapper lookup requires executing a separate scan for each of the 3 declared snapshots (v through $v+2$). The scan for v starts at the first mapping in node n_1^0, and collects P1, P2, and P3, ignoring two repeated mappings to P1. The scan for $v+1$ starts at the second mapping in n_1^0, but otherwise follows the same path. The same is true for $v+2$. The work done by Mapper lookup to scan for $v+1$ and $v+2$ is done by the scan for v. The goal is to eliminate this redundant work while still collecting the mappings needed for each snapshot in the range.

Joint Skippy Scan. A single Mapper lookup scan is insufficient to collect mappings for an arbitrary snapshot range, because mappings needed by a snapshot may not always be copied up to Skippy levels. For example, consider the range from snapshot $v+2$ to $v+4$ depicted in figure 2. A Skippy scan beginning in node n_1^0 will follow the uplink to node n_1^1, and will not collect the correct mapping to P1 needed for $v+4$ (notice that the mapping to P1 pointed to by the start pointer for $v+4$ is not copied up to node n_0^1). The *joint Skippy scan* solves this problem by first executing a *mapLog* scan between the *mapLog* positions pointed to by the start pointers for the first and last snapshots in the range, then executing a regular Skippy scan starting with the last snapshot in the range. For example, if the range is from v to $v+5$, then nodes n_1^0 through n_3^0 will be scanned sequentially, ignoring uplinks; then, after the first mapping in $v+5$ is encountered in n_3^0, the Skippy scan will follow uplinks.

The joint Skippy scan can be seen as joining multiple *mapLog* scans together so that they share the work to collect shared mappings. Because a *mapLog* scan collects all the FEMs needed by a snapshot (see section 3), the portion of the joint Skippy scan that only scans *mapLog* will collect the FEMs written to *mapLog* within the specified range. Any FEM missing during this scan will be encountered during a Skippy scan starting with the last snapshot in the range, by the construction of Skippy. Therefore, a joint Skippy scan will collect all FEMs for a range of snapshots in one scan, effectively merging together multiple Mapper lookup scans. Section 5.1.2 considers the cost of a joint Skippy scan.

5. PERFORMANCE

This section examines Skippy performance analytically and experimentally. Skippy improves the performance of Mapper lookup by copying some mappings to one or more Skippy levels, which comes at the cost of increasing the amount of disk i/o during creation. Our analysis answers two questions. First, how do the updating characteristics of the workload (the skew) effect Skippy benefits and costs? Second, how to select the best configuration for Skippy in practice. We confirm the analytical results of Skippy benefits in a BDB prototype, and in the SNAP system.

5.1 Analysis

We analyze the performance envelope of Skippy using a simplified workload model that directly addresses the issue of skew. To address the worst case cost, we assume that a snapshot is taken after every transaction, which corresponds to continuous data protection (CDP). We use a simple, standard hot/cold updating workload model. We believe that this captures the effect of skew, while providing a framework for a tractable analysis.

Overwrite Cycle Length. The overwrite cycle is the number of page updates that execute before all pages in the database have been modified at least once; the number of updates in the overwrite cycle increases with the chance of modifying the same page more than once (due to skew). Finding the number of updates in the overwrite cycle is equivalent to the well-explored coupon-collector's waiting time problem [15]. The overwrite cycle in a database with n pages and no skew can be approximated as $n * ln(n)$, where the logarithmic factor is due to random selection of already-modified pages. For CDP, the number of mappings

in *mapLog* corresponds to the number of page updates in the overwrite cycle, so we use "overwrite cycle length" to refer to both page updates in and mappings written during the overwrite cycle.

The hot and cold pages are disjoint sets, and so are collected independently, but because the hot section is smaller and modified more frequently, redundant mappings will be contributed disproportionately from the hot section, while the total overwrite cycle length will be dominated by the time to complete an overwrite cycle in the cold section. The length of the overwrite cycle will grow inversely with the probability of a transaction executing in the cold section. Because the hot section is both small and frequently updated, hot pages will have many repeated mappings in an overwrite cycle.

Acceleration. *Acceleration* characterizes the performance benefit that can be derived from Skippy. Acceleration is the ratio of mappings in Skippy level h to mappings in level $h-1$. Mappings which are repeated within a node are not copied up to the current node at the next level, so acceleration improves as the number of redundant mappings within a node increases. As skew increases, nodes become dominated by mappings written from the hot section. Because of this effect, Skippy acceleration is improved when skew increases, effectively combating the increase in the number of mappings in an overwrite cycle due to repeated modifications in the hot section.

Because repeated mappings can only be eliminated within a node, acceleration is proportional to the node size. For skewed workloads, the "blow-up" in the number of mappings is due to the repeated modifications in the hot section even after all the hot pages have been modified at least once, because an overwrite cycle in the cold section has not finished. So, most of the acceleration comes from eliminating repeated mappings created from modifications in the hot section. As figure 3 illustrates, less-skewed workloads do not have as much potential for acceleration as more-skewed workloads.

5.1.1 Cost of Skippy Scan

The cost to use Mapper lookup to construct an SPT using only *mapLog* is the cost of sequentially reading all the mappings in the overwrite cycle beginning with the mapping pointed to by *Start*. The cost to construct an SPT with a 2-level Skippy (a *mapLog* and one Skippy level) is the cost to read a node at the *mapLog* level, the cost of a seek to jump to the Skippy level, plus the cost to scan remaining mappings in the overwrite cycle at the next level. Because of acceleration, the overwrite cycle comprises fewer mappings in the Skippy level, and so is a shorter scan.

Figure 3 depicts SPT construction times for varying workload skews with different Skippy heights by iteratively calculating these quantities based on our analysis of the overwrite cycle length and acceleration factors, and assuming a sequential read speed of $0.04ms$ per $8KB$ page and an average seek time of $8.9ms$. A strong performance benefit is achieved with just a few Skippy levels, with diminishing returns exhibited as more levels are added. This is because, for the workloads in our analysis, the first few levels eliminate most of the redundant mappings created due to skew, and the remaining scan cost is dominated by the cost of reading $n * ln(n)$ mappings to collect all n FEMs in the overwrite cycle.

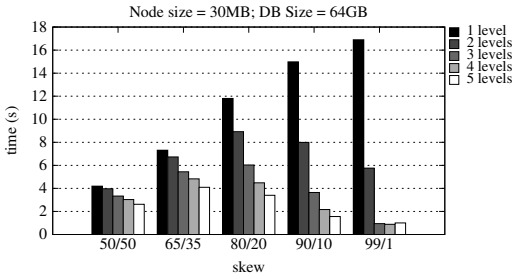

Figure 3: Construction times for various workloads

Our analysis shows that the cost to construct SPT with Skippy can be made similar for each of the workloads we consider, but the number of levels required to converge to this cost in less-skewed workloads can be high while the benefit is relatively low. For example, constructing an SPT for the "50/50" workload and the "90/10" workload shown in figure 3 can both be made to converge to about the same cost (just under 2 seconds), but it would require an additional 5 levels beyond those shown in figure 3 for "50/50" (for a total of 10 levels to achieve it), with a very small benefit. "90/10" converges to this cost after only 5 levels, with a greater overall benefit as compared to not employing Skippy. Skippy is designed to eliminate the impact of *skew* on the cost to construct an SPT, and so for the remainder of the discussion we consider the *optimal* number of Skippy levels to be that which can eliminate the impact of skew (so that the cost is similar to the "50/50" workload lookup cost with no Skippy levels).

Database size. Figure 4 shows the number of Skippy levels needed to achieve similar performance benefit (computed as cost savings over using just *mapLog*) for "90/10" as the size of the database increases and the node size remains fixed. By adding additional levels, performance benefits can be recovered if the database grows.

For example, a 16GB database receives nearly maximum benefit with just 2 Skippy levels; but, if the database size increases to 128GB then the benefit drops from 90% to 70%. The performance improvements given to the 16GB database can also be given to the 128GB database by adding 2 additional Skippy levels, increasing the total number of levels to 4. However, while our analysis shows that adding levels can recover performance, the cost of Skippy creation (figure 5) may make adding more levels impractical for some real-world system configurations. We consider how to scale the node size instead of Skippy height to recover performance when database size increases in the experimental evaluation (table 2).

5.1.2 Cost of Joint Skippy Scan

We know by invariant I_{mapLog} that the mappings written into *mapLog* between the start position of v and the start position of $v+1$ are FEMs for v, and so on for each pair of subsequent snapshots in an arbitrary range of snapshots. Thus we have corollary C_{mapLog}: *all mappings in mapLog between the start position of v and the start position of $v+k$ are FEMs for for one or more of the snapshots in the range v through $v+k-1$.*

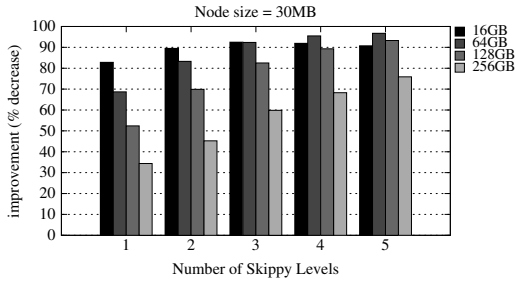

Figure 4: Construction benefit for "90/10"

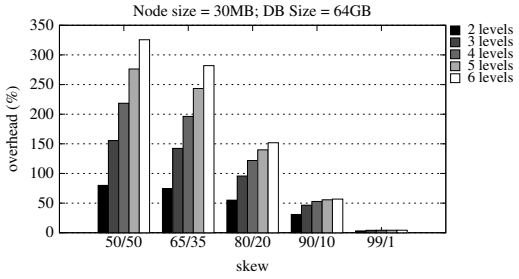

Figure 5: Creation overhead

Because all mappings scanned during the *mapLog* phase of the joint Skippy scan are FEMs, there is no wasted work during this phase. Because all i/o when scanning *mapLog* is sequential, this phase has minimal i/o cost. Thus, by corollary C_{mapLog}, the cost of the joint Skippy scan is the minimal cost to read all FEMs recorded between the declaration of the first and last snapshots in the range, plus the cost of a single Skippy scan to find the FEMs for the last snapshot in the range.

5.1.3 Creation Cost

The cost of writing mappings into *mapLog* with no Skippy levels is low since *mapLog* is an append-only, sequential store, and because Skippy can be written in the background. Mappings are flushed as a batch from large write buffers to amortize seek costs. Skippy levels are structured like *mapLog*, so their cost is likewise similar, although by construction the number of mappings written into each level is different. The cost of writing Skippy is thus the sum of the costs of writing mappings sequentially at each level. In figure 5, we calculate the overhead of creating Skippy as the percent above the baseline cost of writing mappings into *mapLog* by iteratively determining the number of mappings that would be written into each level (the figure does not depict a 1-level Skippy, which is just *mapLog*). As workload skew increases, the number of repeated mappings within a node increases, which decreases the number of mappings copied up from *mapLog* into Skippy levels. Thus, the overhead of creating Skippy decreases as skew increases.

5.1.4 Configuring Skippy

Each additional Skippy level adds creation overhead, while skewed workloads tend to exhibit diminishing returns for Skippy levels beyond 3 or 4. Keeping in mind that Skippy must be non-disruptive as well as enable efficient Mapper lookup, we consider how to configure and tune Skippy.

Databases are often comprised of both mutable and immutable data. The mutable portion may be much smaller than the overall database size (for example, a database may contain immutable video files and a mutable table containing user comments). Skippy need only to be configured to efficiently index mappings for the mutable pages of the database.

Practical Skippy Height. We observed that much of the benefit of Skippy in counteracting workload skew is achieved in the first few Skippy levels. To better quantify this idea, we picked a target cost of 5 seconds to construct an SPT for a 64GB database, then employed exhaustive search to determine the smallest Skippy height that would achieve this performance level. We found that height of 3 was sufficient to achieve this performance for "90/10" and "99/1", while the less skewed "65/35" and "80/20" required 4 levels. This supports the intuition that Skippy provides strong performance benefits to skewed workloads even with just a few levels, imposing minimal performance overhead.

Dynamic Reconfiguration. In practice, a beneficial Skippy height could be configured by keeping statistics that track the acceleration between levels during creation (i.e., the number of FEMs copied between levels). Skippy height can be adjusted dynamically, so that if the acceleration is observed to be poor above level $h_{practical}$, mappings do not need to be copied to higher levels, and the links written at the end of new nodes in level $h_{practical}$ are in-links instead of uplinks. We have designed but have not yet implemented this feature.

Cold Spots. The common skewed workload model assumes a small hot section (a "hot spot"). One could construct a workload that has a small cold section instead of a small hot section. For example, updates could be uniformly random in most of the database except a small subset of very cold pages. In such a workload, the dynamic reconfiguration mechanism might inhibit writing Skippy levels, yet the overwrite cycle could still be quite long. In practice, the overwrite cycles created by such a workload can be forcibly shortened by periodically copying very cold pages to *snapStore* (once per desired overwrite cycle) and recording a mapping to them in *mapLog*. A scan of *lastRetainer* can identify those pages which are cold as indicated by last being retained for a very old snapshot. The age at which a pre-state is forcibly created is a tunable parameter that trades some additional background i/o to limit overwrite cycle length in workloads with small cold sections.

Memory requirements. For Skippy, the tail node of *mapLog* and each Skippy level must be resident in memory. This is necessary to make the check to see if a mapping is repeated within the current node efficient. Our analysis shows that the performance of Skippy can scale if the size of a node scales in proportion to the number of hot pages. Figure 4 shows that for less-skewed workloads with larger hot sections, the effect of increasing database size (and so the number of hot pages by our definition of skew) can have a significant impact on performance if the node size is not also scaled. This is similar to the scaling required to maintain a hit ratio in a read cache. Mappings are quite small, so we believe that our approach is practical even for databases

with large amounts of mutable data. Section 5.3.2 discusses node sizes in our implementation.

The other memory cost in our system is the *lastRetainer* structure. It has a a cost similar to the database page table. Like a page table, and unlike nodes in the Skippy structure, *lastRetainer* can be paged to disk if it is too large to fit comfortably in main memory. We assumed in our analysis that *lastRetainer* fits entirely in memory, but since the access locality in *lastRetainer* will be the same as the locality for database pages, the trade-off of keeping part of the page table on disk is well-understood.

5.2 Evaluation in SNAP

We have implemented Skippy Mapper in the SNAP [21] split snapshot system. We grafted Skippy onto the original SNAP system [21] that writes snapshot page mappings in the *mapLog* and accelerates sequential *mapLog* scan by retaining in-memory checkpoints for selected snapshots.

We used our prototype to conduct an experiment to gauge the impact of Skippy in a running system. The analysis in Section 5 evaluated the overhead of Skippy in terms of the total extra snapshot page table mapping disk i/o required for Skippy creation in an overwrite cycle. In a running storage system, Skippy is created incrementally, in the background, as part of the copy-on-write snapshot creation process that accompanies the database update process, and as such, could slow down the update process, ultimately impacting the foreground application transactions. A useful practical measure of Skippy efficiency is the impact of its background creation cost on the update process. Our experiment gauges the overhead of Skippy on the update process in the prototype system by measuring and comparing the cost of updating a single database page in a system with and without snapshots, and breaking down the overhead due to snapshots into two components, the overhead due to the copying of snapshot pages, and the overhead due to the writing of snapshot page table mappings. We then consider how the overhead with Skippy compares to the overhead of a system without it.

The experiment runs in a client/server system with the Skippy-enhanced SNAP system running at the server. The client generates an application transaction workload by running a variant of the OO7 benchmark [2] read/write traversals T2a declaring a snapshot after each transaction (highest frequency). The application-level benchmark does not allow us to control the page-level update skew directly, typical for an application written in a high-level language. Instead, the benchmark updates randomly selected objects [21]. The resulting workload has a complex page-level behavior but exhibits the following observed page update characteristics. It generates high overwriting, and the randomly-selected object updates leave some pages unmodified for a long time, producing a highly skewed workload (long overwriting cycle), and also stressing the total archiving system overhead.

We do not detail further the experimental setup for brevity. Instead we note that an identically configured experiment in SNAP using in-memory acceleration has shown that even for high-frequency snapshots, the entire split snapshot overhead is low [21] and the cost of writing the snapshot page table mappings is minimal. Our experiment confirms the findings for the Skippy-based system (that accelerates lookup at the cost of extra writing of snapshot page table mappings). A Skippy graft configured for $h = 2$ with Skippy node size set to 512KB contributes 1.3% of the total archiving overhead, out of which 0.3% is due to the additional Skippy levels, with the remainder due to the base housekeeping costs in the entire Mapper subsystem. Based on our measurement and analysis, we therefore conservatively estimate that in practice the cost of writing snapshot page table mappings in the few additional levels required to achieve the, close to optimal, predicted Skippy benefit will remain small.

We used the *snapStore* created by our workload (16K snapshots, 60GB *snapStore*) to run Skippy scans on snapshots with completed overwrite cycles, representing long-lived snapshots that can not take advantage of in-memory meta-data acceleration. The measured Skippy scan costs for reading and scanning a Skippy node were 55ms. Given the workload, compared to a SNAP system without lookup acceleration, the $h = 2$ level Skippy reduced the longest observed overwrite cycle from the OO7 workload by 19-fold. The results indicate the predicted performance benefits of even small Skippy structure in skewed workloads. Our experimental evaluation of a new system, SkippyBDB, examines the average benefit of Skippy using synthetic workloads modeled after our analysis 5.3.

5.3 Evaluation in SkippyBDB

In order to support analytical results with the on-demand Mapper lookup protocol under a deterministic workload, and to gain experience implementing Skippy in a commercial database storage manager, we implemented SkippyBDB, a split-snapshot system built inside Berkeley DB [10] (BDB). SkippyBDB augments the BDB page cache, storing prestates of pages into snapStore and implementing the Mapper creation algorithm. Applications requesting BITE use an unmodified BDB interface for accessing records; the SkippyBDB-enhanced page cache transparently loads snapshot pages from snapStore. We measure the time to build an SPT, which requires a lookup scan over one overwrite cycle, and compare results to analytical expectations.

5.3.1 Experimental Setup

We employ SkippyBDB to measure the performance of Skippy. To examine the performance impact of creating the Skippy index, we ran the following workload: a reader thread executes 20 sequential scans of the entire database; concurrently, a writer thread updates a random item in the database. The writer makes updates until the reader has finished. To conservatively measure the cost of Skippy, we do not skew the workload, thereby maximizing the overhead due to Skippy levels (figure 5). We also employ 5 levels, which would normally not be necessary for an unskewed workload; but, adding many levels conservatively increases the work required to create Skippy. Because Skippy is written during database checkpoints, we ensure that each run is long enough to capture multiple checkpoints. To isolate the cost of Skippy, we modify our implementation to not write snapshot pages, thus eliminating snapstore I/O from our measurements.

To benchmark the benefit of Skippy when constructing SPTs, we first create an overwrite cycle as follows: a writer makes successive random updates, declaring a snapshot after each update, until each page has been modified at least once. We generate the synthetic workload used in the analysis by *skewing* more transactions to the hot section as the

Table 1: Time to construct SPT for various skews

Skew	Skippy Height	Time (s)
50/50	1	**13.8**
80/20	1	19.0
	2	15.8
	3	14.7
	4	**13.9**
99/1	1	33.3
	2	**6.69**

hot section shrinks in size. To test the benefit of Skippy, we measure the time it takes to construct the SPT by scanning from the first to last mapping in the overwrite cycle (time includes CPU cost of inserting mappings into page table, which is implemented as a hash table). Since our benchmark of Skippy benefit is only concerned with mapLog and the Skippy index, in the interest of time we simulate the actual workload by selecting random page numbers instead of by updating an actual database. We compared a sample of our simulated *mapLog* and Skippies with those from our full BDB prototype and found them to be similar.

Hardware. All measurements are taken on a Dell PowerEdge with dual 2.80GHz processors (only one of which is used for the experiment), 4GB of physical RAM, and two Seagate SCSI drives (model ST3146755SS) on the same SCSI bus (only one of which is used during the experiment). The test machine is running Debian Etch with the x86_64 build of Linux 2.6.22. Berkeley DB is hosted on an ext3 file system. Berkeley DB defaulted to a page size of $4K$, which we did not change.

5.3.2 Results

Creation Cost. We averaged the time for the reader to complete 20 sequential reads of a $1.3GB$ database with a $100M$ cache over 5 runs for each of two cases: the first in which the writer takes a snapshot after each transaction, the second in which the writer takes no snapshots. The second case incurs no overhead from Skippy since no mappings will be created. We ignored the time for the first sequential read from each run to minimize impact of a cold cache (our test database is organized as a BTree so inner nodes are cached after the first iteration). Both runs included 9 checkpoints. We compared these two cases and found no significant impact on the time to complete the scan from the creation of Skippy. Since Skippy is written to a separate disk from the database, writing *mapLog* and Skippy is not disruptive to reader I/O done in the current state.

Skippy Benefit. Table 1 shows the cost of constructing an SPT for one overwrite cycle for various representative skews and Skippy heights in a 100M database and a 50K node (a node of this size holds 2560 mappings, which are each 20 bytes, which is $1/10^{th}$ the number of pages in the database). Mappings could be decreased in size with encoding of snapshot page address at no scan-time cost, increasing node capacity. Each measurement is the average of 3 experimental runs, with negligible variance from the mean. For a node size that is small compared to the number of pages in the database, we observe that the "50/50" workload achieves minimal benefit from Skippy. This is because there is no skew, and the likelihood of repeating mappings within a node is relatively small. The cost of a *mapLog* scan for the "50/50" workload is essentially the cost due to the $n * ln(n)$ blow-up.

Table 2: Scaling configuration with database size

Database Size	Node Size	Skippy Benefit
100M	50K	27%
400M	50K	8%
	200K	31%

We expect from the analysis that for a skewed workload, Skippy will be able to reduce the scan time to be as small as the scan for "50/50" (figure 3). Indeed, the "80/20" workload required 3 Skippy levels above *mapLog* to achieve a Skippy scan time similar to a *mapLog* scan in "50/50". Table 1 also shows that for a highly skewed workload like "99/1" for which the node size is actually larger than the number of hot pages, Skippy can easily accelerate the scan to be faster than the "50/50" *mapLog* scan. We observed similar behavior for the other workloads studied in the analysis.

Table 2 shows that, as predicted by the analysis, increasing the database size without also scaling the node size can decrease Skippy performance. We measured two database sizes (100M and 400M). If the node size is not scaled with the database, then Skippy acceleration is decreased; however, by scaling the node we achieve similar benefit. Because mappings are small, node sizes can remain practical even as databases grow large. For example, this scaling factor would require a 5M node for a 10G database. Note that for our workload, the hot section always scales with database size, for real workloads the hot section may not change size as the entire database grows, lessening the need to scale the node size. Additionally the random updating workload used in our analysis and experiment does not exhibit temporal locality. Since a node is comprised of mappings from contemporaneous transactions, temporal locality would increase the number of repeated mappings within a given node. In this sense, the workload we test is the worst-case for Skippy. We expect that Skippy will perform as well or better with workloads that exhibit temporal locality than with our synthetic workload.

We have also experimentally confirmed that the length of an overwrite cycle is not a function of snapshot age. Because the length of an overwrite cycle is determined by the random order in which mappings are generated, two randomly-selected overwrite cycles generated by the same workload may vary in length. We created 1,000,000 snapshots using the technique described above, and then continued adding mappings to *mapLog* until 1000 evenly-spaced snapshots selected from this set had completed overwrite cycles. This created 1000 snapshots with varying ages distributed inside the *mapLog*. We averaged the number of mappings in each of the 1000 snapshots and found that the standard deviation was at least an order of magnitude smaller than the mean for both unskewed ("50/50") and skewed ("90/10") workloads.

6. RELATED WORK

Work on multiversion data has been carried out in both databases and file systems. For both databases and file systems, most work on versioned data has taken the copy-on-write, no-overwrite update approach. To our best knowl-

edge, Skippy is the first indexing approach that solves the multiversion access problem in a copy-on-write system with update in-place. A poster describing the Skippy technique appeared in [19].

Skippy departs from existing database proposals for multiversion data in several important ways: the level of snapshots, the assumed database update architecture, and the acceleration method. All existing database multiversion access proposals, that we know of, operate at the logical database level, indexing logical records, and targeting specific types of queries. For example, Snapshot Index [25] targets snapshot queries ("find all records in version v"), Time-Split B-tree (TSB) [9] and the Multiversion B-tree (MVBT) [1] target key-range queries ("find records in version v within range r"). In contrast, Skippy operates at a lower level (the storage manager), indexing disk page-level snapshots. As such, Skippy is agnostic to what type of code accesses the snapshot. This way, all read-only queries supported by the database (and all the application programs) will run in the snapshot, incurring the same number of page accesses, at a page access cost similar to a secondary index.

Regarding data update, existing multiversion database proposals transform record updates into inserts [8, 25, 11, 24]. The previous record version remains in place, and the past version index structure overtakes the current access structure, updating the index to provide access to the current version. Since the index structure is shared by the past and the current state, access to the past state inherits the efficient ordered set access properties from the current state. However, existing temporal database indexing approaches (e.g., TSB) must periodically reorganize data to maintain current-state clustering, incurring additional index updates. In split snapshots, on the other hand, an update to a record copies a page (the pre-state) into *snapStore* and appends a mapping to *mapLog*, maintaining clustering of records and minimizing interference with the current state.

Because of the architectural differences, a direct performance comparison between Skippy and the existing multiversion indexing proposals is difficult. To put our work in perspective, we consider how history length impacts both Skippy and Time Split B-tree (TSB) [9], a state-of-the-art multiversion index designed for ImmortalDB [8]. TSB is a record access method that combines the key and time dimensions into a single B-tree, enabling efficient "as-of" queries similar to BITE in Skippy.

The search cost for each record in TSB increases logarithmically with the size of history. In contrast, our approach requires a scan of unsorted mappings, but that scan length is dependent only on database size and workload, and is independent of history size. An individual record lookup in TSB will be faster than a Skippy scan (unless the size of the TSB is quite huge). This is why a scan produces an SPT, a sorted lookup table containing a mapping for each page in the database (at time of snapshot). The cost of the scan can be amortized over all lookups in the SPT after it is built, making the Skippy approach very attractive for workloads which will execute many back-in-time operations on the same snapshot. Joint scans for ATE queries provide additional amortization compared to equivalent search tree lookups.

Some commercial databases store consistent page-level snapshots separately to avoid inteference with current state and provide quick recovery from errors. Oracle's Flashback [14] retains snapshots at the page level in a size-limited store, augmented with archived transaction logs [14] to enable snapshot roll-back to the needed state. Microsoft's VSS service [17] service allows applications to take consistent page-level snapshots efficiently by coordinating the flushing of application buffers. These approaches target short-lived snapshots, but we think that Skippy could naturally complement them making it possible to access long-lived snapshots.

The Network Appliance Filer, a widely used, high-performance storage system, provides snapshots through the WAFL [6] file system. WAFL uses no-overwrite updates for current-state storage (instead of updating modified blocks in-place as in the split snapshot approach), and so declusters current state data. Unlike Skippy-indexed snapshots, WAFL-based snapshots are not long-lived.

The ext3cow versioning file system [12] provides snapshots of the entire file system by modifying ext3 to support no-overwrite copy-on-write updates for data pages. ext3cow provides fast current state meta-data access by separating past and present meta-data and updating current meta-data in-place, but it employs version chaining so, unlike Skippy, access times are proportional to history length.

CVFS [23] is a high-performance, continuous-versioning system for a no-overwrite log-structured file system, supporting intrusion analysis. It logs versioned file meta-data similarly to the *mapLog* in Skippy, but accelerates temporal queries using TSB [9].

Skippy bears superficial resemblance to Skip Lists [13]. Both Skippy and Skip Lists are concerned with constructing "fast-lanes" for searching. However, Skip Lists are an in-memory data structure that maintains a sorted set with probabilistic forward links, while Skippy is composed of a hierarchy of unordered sets (of mappings) optimized for sequential disk i/o. Additionally, while Skip Lists accelerate the lookup for a single item, the Skippy index accelerates the search for a subset of items (the FEMs needed to construct an SPT).

7. CONCLUSION

Decreasing disk costs make it possible for a database to store on-line snapshots of past states for long periods of time but up to now there was no satisfactory way for a general purpose database to run code on-line over long-lived snapshots. We have presented Skippy, the first efficient solution to the problem of indexing long-lived split copy-on-write snapshots for on-line access by BITE and ATE programs.

We have designed Skippy for a split snapshot system with a sequential snapshot store but the approach is more general and we believe can be adapted to different snapshot store organizations and different current state storage systems. For example, Skippy could be used with a content addressable past state organization. Such approach would simply store a snapshot page content hash instead of snapshot store address in the snapshot page tables and *mapLog* mappings, and require that the snapshot system enforce the invariant I_{mapLog}.

Skippy-based long-lived split snapshots potentially could be adapted to become a standard database system feature bringing ad-hoc on-line past state analysis to general applications. Such approach is attractive because it is consider-

ably simpler and more general than the alternative of capturing past states and providing temporal access methods at the logical database level. Our BDB prototype demonstrates a step in this direction.

8. ACKNOWLEDGEMENTS

We wish to thank our shepherd, David Dewitt, and our anonymous reviewers for their invaluable feedback. Sam Madden's database research group at MIT and Michael Stonebraker provided helpful comments. Finally, we thank Ioana Manolescu and the anonymous repeatability verifiers for going the extra mile interpreting our results. This research was partially supported by NSF grant CNS-0427408.

9. REPEATABILITY ASSESSMENT RESULT

Figure 3 and tables 1 and 2 have been verified by the SIGMOD repeatability committee.

10. REFERENCES

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal: Very Large Data Bases*, 5(4):264–275, 1996.

[2] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The oo7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington D.C., May 1993.

[3] S. Chandrasekaran and M. Franklin. Remembrance of stream past: Overload-sensitive management of archived stream. In *VLDB 2004*, Toronto, Canada, 2004.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, USA, December 2004.

[5] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[6] D. Hitz, J. Lau, and M. Malcom. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, CA, January 1994.

[7] K. J. Jacob and D. Shasha. Fintime: a financial time series benchmark. *SIGMOD Rec.*, 28(4):42–48, 1999.

[8] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal db: transaction time support for sql server. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 939–941, 2005.

[9] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1990. ACM Press.

[10] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the 1999 Summer Usenix Technical Conference*, Monterey, California, June 1999.

[11] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *Knowledge and Data Engineering*, 7(4), 1995.

[12] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system, 2003.

[13] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.

[14] A. Romero and L. Ashdown. *Oracle Database Backup and Recovery Basics*, chapter Oracle Flashback Technology: Alternatives to Point-in-Time Recovery. Oracle Corporation, Redwood Shores, CA, 10g release 2 (10.2) edition, 2005.

[15] S. M. Ross. *Probability Models for Computer Science*, chapter Martingales. Harcourt Academic Press, San Diego, first edition, 2002.

[16] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2), 1999.

[17] A. Sankaran, K. Guinn, and D. Nguyen. Volume shadow copy service. *Power Solutions*, March 2004.

[18] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Otir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, 1999.

[19] R. Shaull, L. Shrira, and H. Xu. Skippy: Enabling long-lived snapshots of the long-lived past. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, Cancun, Mexico, 2008. IEEE Computer Society.

[20] L. Shrira, C. van Ingen, and R. Shaull. Time travel in the virtualized past: Cheap fares and first class seats. *Haifa Systems and Storage Conference*, SYSTOR 2007.

[21] L. Shrira and H. Xu. 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.

[22] L. Shrira and H. Xu. Thresher: An efficient storage manager for copy-on-write snapshots. In *USENIX '06: Proceedings*, Berkeley, CA, USA, 2006. Advanced Computer Systems Association.

[23] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003. USENIX Association.

[24] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th International Conference on Very-Large Data Bases*, Brighton, England, UK, September 1987.

[25] V. J. Tsotras and N. Kangelaris. The snapshot index: an i/o-optimal access method for timeslice queries. *Information Systems*, 20(3), 1995.