# Skippy: Enabling Long-Lived Snapshots of the Long-Lived Past

Ross Shaull, Liuba Shrira, Hao Xu

*Department of Computer Science, Brandeis University*
*Waltham, Massachusetts, USA*
{rshaull,liuba,hxu}@cs.brandeis.edu

## Abstract

Decreasing disk costs have made it practical to retain long-lived snapshots, enabling new applications that analyze past states and infer about future states. Current approaches offer no satisfactory way to organize long-lived snapshots because they disrupt the database in either short or long run. Split snapshots are a recent approach that overcomes some of the limitations. An unsolved problem has been how to support efficient application code access to arbitrarily long-lived snapshots. We describe Skippy, a new approach that solves this problem. Performance evaluation of Skippy, based on theoretical analysis and experimental measurements, indicates that the new approach is effective and efficient.

## I. Introduction

The storage manager of a general-purpose database system can retain consistent disk page level snapshots and run application code "back-in-time" against arbitrarily long-lived past states, which are virtualized in the page buffer to operate like the current state (including indices and other metadata pages). This opens the possibility that functions, such as analysis and audit, formerly available in specialized temporal databases, can become available to applications in general-purpose databases. Until recently, a general purpose database system that updates data in-place had no efficient way to support on-line, back-in-time execution (BITE) over snapshots. Retaining frequent long-lived snapshots was simply too disruptive to the database performance [1], and BITE over long-lived copy-on-write snapshots has been prohibitively slow.

Split snapshots [2], [3], [4] is a recent approach that overcomes some of the limitations of earlier systems and sets the context for our work. This approach provides snapshots in a form that virtualizes the database storage, adding a layer of indirection between the physical address of disk pages and the database paging architecture similar to shadow tables [5], allowing a database to run unmodified application programs and access methods over consistent snapshots in real-time. 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, allowing the database to support update-in-place.

## II. Long-lived Split Snapshots

In a split snapshot system [2], [3], [4], an application takes a transactionally-consistent snapshot by issuing a snapshot *declaration request* and receiving a *snapshot name* from the system. Consider a database storage system that includes a set of disk pages $P1$, $P2$, ..., $Pk$. A snapshot consists of a set of *snapshot pages* and a *snapshot page table* (SPT) that maps snapshot pages (pre-states) to their disk locations. Pre-states are stored in a log-structured component called *snapStore*, and created using copy-on-write. Pre-states can be found by scanning forwards from the first page retained for the snapshot, storing in the SPT the first entry for each page. Scanning entire pages is expensive; SNAP [2] reduces the cost by a large constant factor by replacing the scan of pages with a scan of a log of pointers (called mappings) to pre-states.

The problem is that when some pages are modified much more frequently than others (from workload *skew*), many hot pages may be overwritten before the first modification to cold page $P$, so a scan to find $P$ has to pass over large number of repeated mappings that correspond to the same small set of hot pages. The SNAP system addresses this problem by keeping the mappings of infrequently modified pages in memory. This solution, however, does not scale as snapshot lifetimes increase.

*Mapper* is the split snapshot system component that tracks the location of pre-states. 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 $v$. Call such a pre-state a *page retained for* $v$. Without constraining the snapshot page copying order, the Mapper write function enforces the following invariant: $I_{mapLog}$*: all the mappings for pages retained for snapshot* $v$ *are written before all the mappings for pages retained for snapshot* $v+1$.

Mapper lookup relies on the invariant $I_{mapLog}$ when searching for the mappings. Let $start(v)$ be the first mapping for a page retained for a snapshot $v$. Mapper finds a mapping for a page $P$ in $v$ by sequentially scanning the *mapLog* from the position $start(v)$ onward, returning the first mapping it encounters for a page $P$. $I_{mapLog}$ guarantees that the first encountered mapping (FEM) corresponds to the pre-state that was captured when the page $P$ was modified for the first time after declaration of $v$, and is, therefore, the correct mapping.

## A. *Skippy Mapper*

To support efficient BITE that runs application code on a snapshot by transparently paging in snapshot pages, the entire SPT is constructed when an application requests a snapshot. To do this, the system needs to find in the *mapLog* all the mappings for a given snapshot. The maximum length of the scan to construct $SPT(v)$ is determined by the length of the *overwrite cycle* of a snapshot $v$, defined as the transaction history interval starting with the declaration of $v$, and ending with the transaction that modifies the last database page that has not been modified since the declaration of $v$. *Skippy* accelerates SPT construction for skewed workloads by allowing the lookup scan to skip over the repeated mappings that lengthen the overwrite cycle. FEMs are collected from fixed-size intervals in *mapLog* (called *nodes*) and written into higher-level logs that contain FEMs needed for lookup scans, but fewer repeated mappings. Additional Skippy levels can be introduced by subdividing the previous level into nodes in the same manner. Nodes are terminated with an *uplink* which points to the node at the next level at which the scan should continue (except the top-most level, where nodes are terminated with an *in-link* which points to the next node at the same level). We label nodes as $n_i^l$, where $l$ is the level (with *mapLog* known as level 0) and $i$ is the node index (starting with 0 at each level).

Figure 1 shows an example of a 2-level Skippy resulting from declaring snapshots $v1$ through $v8$ in a skewed workload. $P4$ and $P5$ are retained for $v1$, and again for $v2$. $P1$ is retained for $v3$, then again for $v4$, and so on. Mappings to $P1$ are 3 times as common as mappings to other pages in *mapLog*. Some mappings are FEMs for multiple snapshots (e.g., the mapping to $P1$ retained for $v3$ is also the FEM to $P1$ for $v1$ and $v2$). Consider a Skippy lookup scan to construct $SPT(v1)$ in figure 1. The scan begins in node $n_0^0$ containing $start(v1)$, then continues to the end of $n_0^0$, collecting FEMs for $P4$ and $P5$. The scan then follows the uplink to $n_0^1$, collecting FEMs for $P1$ and $P2$. Finally, the in-link is followed to $n_1^1$ and the FEM for $P3$ is collected. In this example, Skippy allows the scan to avoid 3 repeated mappings for $P1$.

# III. Performance

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 standard hot/cold database model with updates randomly distributed among pages within the hot and cold sections (e.g., an “80/20” workload, in which 80% of the transactions modify one of 20% of of the database pages). This captures the effect of skew, while providing a framework for a tractable analysis.

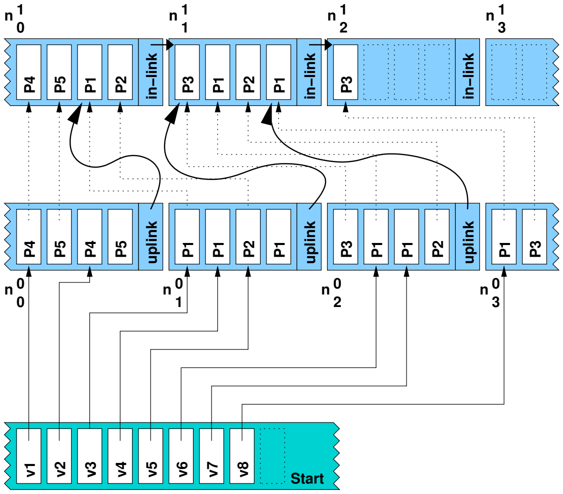

Fig. 1: Example fragment of a 2-level Skippy

## A. *Analysis*

The *overwrite cycle* is the number of transactions that execute before all pages in the database have been modified at least once, and is lengthened as the chance of modifying the same page more than once increases (due to skew). Finding the number of transactions in the overwrite cycle is equivalent to the well-explored coupon-collector’s waiting time problem [6]. 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. As skew increases, the number of cold pages increases, worsening the impact of random page selection on overwrite cycle size. Repeated mappings will be contributed disproportionately from the hot section, while the overwrite cycle length will be determined by the time to complete an overwrite cycle in the cold section. *Acceleration* is the ratio of mappings in an overwrite cycle in Skippy level $h$ to $h-1$. Because repeated mappings can only be eliminated within a node, acceleration improves as node size increases. Because most repeated mappings are contributed from the hot section, most of the acceleration comes from reducing the impact of skew.

Figure 2 depicts construction times for varying workloads 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 0.04$ms$ per 8$KB$ page and an average seek time of 8.9$ms$. A strong performance benefit from acceleration is achieved with just a few Skippy levels, with diminishing returns exhibited as more levels are added. As expected, the less-skewed workloads are less expensive when a Skippy scan is not employed, and receive less benefit from Skippy, since they have fewer repeated mappings due to skew.

## B. *Experimental Evaluation*

In order to support analytical results with the Mapper lookup protocol under a deterministic workload, and to gain experi-

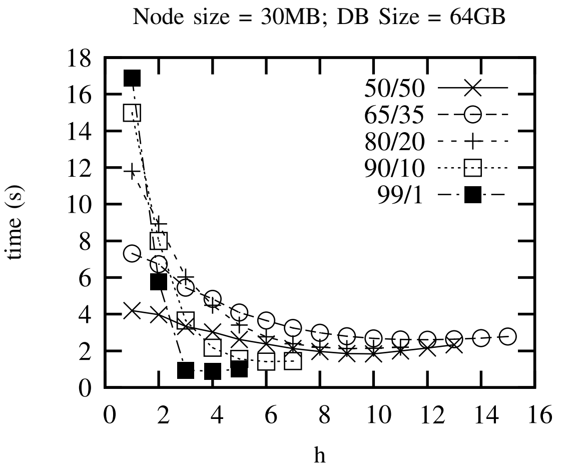

Fig. 2: Construction times for various workloads

TABLE I: Time to construct SPT for various skews

| Skew | Skippy Height | Time (s) |
|---|---|---|
| 50/50 | 0 | **13.8** |
| 80/20 | 0 | 19.0 |
| | 1 | 15.8 |
| | 2 | 14.7 |
| | 3 | **13.9** |
| 99/1 | 0 | 33.3 |
| | 1 | **6.69** |

ence implementing Skippy in a database storage manager, we implemented SkippyBDB, a split-snapshot system built inside Berkeley DB [7] (BDB). We measure the time to build an an SPT, which requires a lookup scan over one overwrite cycle, and compare results to analytical expectations. We timed construction of an SPT for various workloads (including CPU cost of inserting mappings into the page table, implemented as a hash table). For expediency we chose to simulate the actual workload by selecting random page numbers (with *skewed* distribution) for each mapping and directly calling the Mapper write method. All measurements are taken on a Dell PowerEdge using one 2.80GHz processor, 4GB of RAM, and one Seagate 15K rpm SAS hard drive, running Debian Etch with the x86_64 build of Linux 2.6.22. BDB is hosted on an ext3 file system. BDB defaulted to a page size of $4K$, which we did not change.

Table I 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 is $1/10^{th}$ the number of pages in the database). Mappings could be decreased in size with encoding of pre-state address at no scan-time cost, increasing node capacity. Each measurement is the average of 3 experimental runs, with negligible variance from the mean. The "50/50" workload achieves minimal benefit from Skippy, so we present only the cost of a *mapLog* scan. We expect from the analysis that for a skewed workload, Skippy will be able to eliminate the effect of skew, reducing scan time to close to the "50/50" cost (figure 2). Indeed, the "80/20" workload required 3 Skippy levels to achieve a Skippy scan time similar to a *mapLog* scan in "50/50". For "99/1", the node size is larger than the number of hot pages, and so Skippy can accelerate the scan to be faster than the "50/50" maplog scan since it is able to eliminate some repeated cold mappings as well.

## IV. Related Work

Running BITE on a snapshot resembles an "as-of" query in a multiversion database [8]. By accelerating BITE, Skippy serves a similar purpose to multiversion access methods that index 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. Like the state-of-the-art multiversion access methods for as-of queries [8], Skippy guarantees that the time to access a snapshot is independent of the snapshot history length and update workload, an important requirement for long-lived snapshots.

## V. Conclusion

Until now there has been no satisfactory way for a general-purpose database to efficiently retain and index page-level split snapshots for real-time code execution. We have presented Skippy, which to the best of our knowledge is the first solution to solve such a problem for an arbitrary number of long-lived snapshots. Skippy-based page-level snapshots could become an attractive standard feature in general purpose database systems, since the approach is signifantly simpler and more general than the alternative of capturing past states at the logical record level.

The Skippy approach could also be generalized to other snapshot page store organizations, since only the sequential order of mappings is constrainted by the invariant $I_{mapLog}$. For example, Skippy could be used with a content-addressable past state organization. Such an approach would store a page content hash instead of a pre-state address in each mapping, and would offer the benefit of de-duplicating past-state pages.

## References

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

[2] 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 (ICDE'05)*. Washington, DC, USA: IEEE Computer Society, 2005.

[3] ——, "Thresher: An efficient storage manager for copy-on-write snapshots," in *USENIX '06: Proceedings*. Berkeley, CA, USA: Advanced Computer Systems Association, 2006.

[4] 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.

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

[6] S. M. Ross, *Probability Models for Computer Science*, 1st ed. San Diego: Harcourt Academic Press, 2002, ch. Martingales.

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

[8] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys*, vol. 31, no. 2, 1999.