Indexing on Modern Hardware: Hekaton and Beyond

Justin Levandoski David Lomet Sudipta Sengupta
Adrian Birka Cristian Diaconu
Microsoft
{justin.levandoski, lomet, sudipta, adbirka, cdiaconu}@microsoft.com

ABSTRACT

Recent OLTP support exploits new techniques, running on modern hardware, to achieve unprecedented performance compared with prior approaches. In SQL Server, the Hekaton main-memory database engine embodies this new OLTP support. Hekaton uses the Bw-tree to achieve its great indexing performance. The Bw-Tree is a latch-free B-tree index that also exploits log-structured storage when used "beyond" Hekaton as a separate key value store. It is designed from the ground up to address two hardware trends: (1) Multi-core and main memory hierarchy: the Bw-tree is completely latch-free, using an atomic compare-and-swap instruction to install state changes on a "page address" mapping table; it performs updates as "deltas" to avoid update-in-place. These improve performance by eliminating thread blocking while improving cache hit ratios. (2) flash storage: the Bw-tree organizes secondary storage in a log-structured manner, using large sequential writes to avoid entirely the adverse performance impact of random writes. We demonstrate the architectural versatility and performance of the Bw-tree in two scenarios: (a) running live within Hekaton and (2) running as a standalone key value store compared to both BerkeleyDB and a state-of-the-art in-memory range index (latch-free skiplists). Using workloads from real-world applications (Microsoft XBox Live Primetime and enterprise deduplication), we show the Bw-tree is 19x faster than BerkeleyDB and 3x faster than skiplists.

1. INTRODUCTION

A B-tree index supports high performance in both access to individual keys and key-sequential access to subranges of keys. This combination of random and range access has made the B-tree the indexing method of choice within both database systems and stand-alone key value stores. The systems for which the B-tree was designed were uni-processors. They used disks for persistent storage. Those days are long gone. Our demonstration showcases the Bw-tree [8], a new B-tree whose design enables very high performance in the new hardware environment. The Bw-tree addresses two important aspects of modern hardware:

Modern Multi-core Processors: Modern computer systems all use multi-core processors to achieve high performance and extensive memory hierarchies to achieve good memory system performance.

1. Exploiting multiple core systems requires increased concurrency. In that setting, latches are more likely to block and limit scalability [1]. The Bw-tree is latch-free, ensuring a thread never yields or even re-directs its activity in the face of conflicts.

2. Multi-core memory performance depends on high cache hit ratios. Updating memory in place results in costly cache invalidations, especially on multi-socket machines. The Bw-tree performs "delta" updates that avoid in-place updates, reducing invalidations and preserving previously cached page data.

Addressing these two aspects leads to great in-memory indexing performance. For this reason, the Bw-tree is the key range index used by the Microsoft SQL Server Hekaton in-memory OLTP engine [4].

Modern Secondary Storage. Hard disk I/O latency and low I/O rate limit indexing performance. Flash storage offers higher I/O operations per second and lower latency at lower cost per I/O. Due to this performance advantage, several recent storage systems (e.g. Amazon's DynamoDB) explicitly exploit flash [5]. The Bw-tree targets flash storage as well. However, flash has performance idiosyncracies that must be addressed. While flash has fast random and sequential reads, it needs an erase cycle prior to write, making random writes slower than sequential writes. While flash SSDs typically have a mapping layer (the FTL) to mask this difficulty, a noticeable slowdown still exists. Experiments have shown even high-end FusionIO drives exhibit a 3x faster performance for sequential writes than for random writes [3]. The Bw-tree performs log structuring itself at its storage layer to avoid a dependency on FTL and to reduce I/O instruction path with batched writes. This makes write performance as high as possible for *both* high-end and low-end flash devices.

Our demonstration showcases the design and performance of the Bw-tree. We describe the novel techniques we designed to achieve in-memory latch-free behavior and log-structuring on storage. We demonstrate performance in two settings: (1) as the range index running end-to-end in Hekaton; and (2) as an independent key value store. Our live

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2594536.

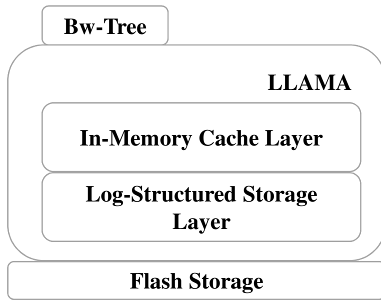


Figure 1: The Bw-tree implemented over LLAMA.

demo runs workloads from two real-world applications: Microsoft's XBox Live Primetime and an enterprise deduplication workload from Windows Server. This demostrates up to a 3x speedup over latch-free skiplists (state-of-the-art in main-memory indexing) and a 19x speedup over the BerkeleyDB B-tree (a traditional storage-based B-tree).

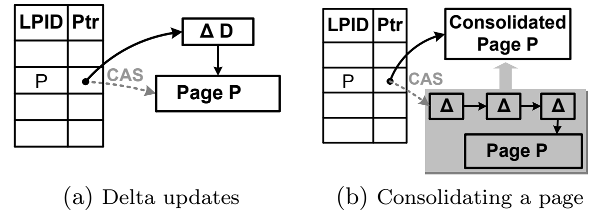
2. THE Bw-TREE DESIGN

The Bw-tree is a B+-tree [2] style index that provides logarithmic access to keyed records from a one-dimensional key range, while providing linear time access to sub-ranges. As depicted in Figure 1, the Bw-tree is built on top of LLAMA [7], our cache and storage subsystem that supports building high-performance page-oriented access methods. LLAMA is divided into layers: (1) an in-memory cache layer serves the Bw-tree with in-memory pages (bringing the page in from flash storage if necessary) and provides latch-free "delta" updates to pages; (2) a storage layer that implements our log-structured store (LSS) over flash. Other access methods can be implemented over LLAMA (e.g. we have implemented linear hashing as well), but for demo purposes we describe our Bw-tree implementation.

The layering in Figure 1 provides an extremely versatile design since (1) it is architecturally compatible with existing database kernels; (2) it is suitable as a stand-alone key-value store; (3) it can be used as an atomic record store in a decoupled transactional system [6]; and (4) due to its latch-freedom, it can serve as an efficient range index in a main-memory database by disabling the LLAMA flash layer (the Bw-tree is the range index in the SQL Server Hekaton main-memory database [4]).

2.1 The Mapping Table

The cache layer of LLAMA maintains a *mapping table*, that maps logical pages to physical pages. Logical pages are identified by a logical "page identifier" or PID. The mapping table translates a PID into either (1) *a flash offset*, the address of a page on stable storage, or (2) *a memory pointer*, the address of the page in main memory. The mapping table is the central location for managing our "paginated" tree. All links between Bw-tree nodes are PIDs, not physical pointers. The mapping table enables the physical location of a Bw-tree node (page) to change on every update and every time a page is written to stable storage, without requiring that the location change propagate to the root of the tree, because inter-node links are PIDs that do not change. This "relocation" tolerance enables both delta updating of the node in main memory and log structuring of our stable storage.



(a) Delta updates (b) Consolidating a page

Figure 2: Delta updates and consolidation.

Bw-tree nodes are thus logical and do not occupy fixed physical locations, either on stable storage or in main memory. This means we have flexibility in how we physically represent nodes. Furthermore, we permit page size to be elastic, meaning we can split pages when convenient as size constraints do not impose a splitting requirement.

2.2 In-Memory Page Updates

Neither readers *nor* writers block when accessing Bw-tree pages in memory due to latch-free page updates in LLAMA. Being latch-free enables us to drive processors to close to 100% utilization. Instead of latches, LLAMA installs state changes using the compare and swap (CAS) instruction[1].

2.2.1 Delta Updating

The Bw-tree updates pages by creating a delta record (describing the change) and prepending it to an existing page state (the delta record contains a pointer to the existing page state). It then requests that the LLAMA cache layer install the (new) memory address of the delta record into the page's slot in the mapping table using a CAS instruction. If the CAS succeeds, the delta record address becomes the new physical "root" address of the page, thus updating the page. This strategy is used both for data changes (e.g., inserting a record) and management changes (e.g., spliting a page). Delta updating simultaneously enables latch-free access in the Bw-tree and preserves processor data caches by avoiding update-in-place. Figure 2(a) depicts a delta update record D prepended to page P; the dashed line represents P's original address, while the solid line to D represents P's new address. We occasionally consolidate pages by creating a new page that applies all delta changes to a search optimized base page. This reduces memory footprint and improves search performance. A consolidated form of the page is also installed with a CAS, as depicted in Figure 2(b) showing the consolidation of page P with its deltas into a new "Consolidated Page P".

2.2.2 Structure Modifications

Index structure modifications operations (SMOs) such as node splits and deletes/merges introduce changes to more than one page. This presents a problem in a latch-free environment since (a) we cannot change multiple pages with a single CAS and (b) we cannot employ latches to protect parts of our index during the SMO. All Bw-tree SMOs are performed in a latch-free manner; to our knowledge this has never been done before. The main idea is to break an SMO into a sequence of atomic actions, each on a single page and

[1]CAS is an atomic instruction that compares a given *old* value to a *current* value at location L, if the values are equal the instruction writes a *new* value to L, replacing *current*.

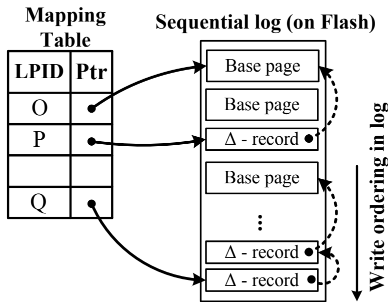


Figure 3: Log structured storage.

installable via a CAS. The details of latch-free SMOs are described elsewhere in our full paper [8].

2.3 Log Structured Store

2.3.1 *Caching*

In addition to latch-free page updates, LLAMA is responsible for reading, flushing, and swapping pages between memory and flash. It maintains the mapping table and provides the abstraction of logical pages to the Bw-tree. Pages in main memory are occasionally written (flushed) to stable storage for a number of reasons. For instance, the Bw-tree may assist in transaction log checkpointing if it is part of a transactional system such as Deuteronomy [6, 9], or to reduce memory usage. Flushing and "swap out" of a page installs a flash offset in the mapping table and permits reclaiming page memory.

2.3.2 *Storage Management*

Log structuring. LLAMA organizes data on flash storage in a log structured manner similar to a log structured file system [12]. Thus, each page flush relocates the position of the page on flash (an additional reason for using our mapping table). Log structured storage (LSS) has the substantial advantage of greatly reducing the number of writes per page and makes the writes "sequential". That is, it converts many random writes into one large multi-page write.

Figure 3 depicts an example of our LSS. A logical page consists of a base page and zero or more delta records reflecting updates to the page (much like its representation in memory). This allows a page to be written to flash in pieces when it is flushed. Thus, a logical page on flash corresponds to records on possibly different physical device blocks that are linked together using file offsets as pointers. Further, a physical block may contain records from multiple pages.

Incremental flushing. To keep track of which part of the page is on stable storage and where it is, we use a *flush delta record*, which is installed by updating the mapping table entry for the page using a CAS. Flush delta records also record which changes to a page have been flushed so that subsequent flushes send *only incremental page changes* to stable storage. This can dramatically reduce how much data is written during a page flush, increasing the number of page updates that that take place during a flush, and hence reducing the number of I/O's per page. There is a penalty on reads, however, as the discontinuous parts of pages must all be in main memory to make a page accessible in the main memory cache. This penalty is mitigated by the very high random read performance of flash.

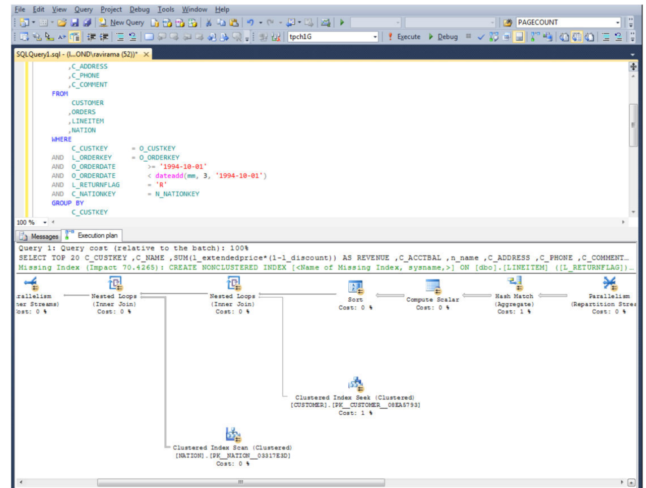


Figure 4: Bw-Tree running in SQL Server Hekaton.

3. DEMONSTRATION SCENARIO

Our demonstration scenario showcases both the architectural versatility and performance of the Bw-tree. First, we demonstrate the Bw-tree running live within the SQL Server Hekaton engine in a real enterprise OLTP scenario. Second, we demonstrate the Bw-tree running as a standalone key-value store and compare its performance to two state-of-the-art range indexing architectures running on workloads from two real-world applications.

3.1 Bw-Tree in SQL Server Hekaton

This demo-scenario showcases the Bw-tree running as a purely in-memory latch-free index within Hekaton [4], Microsoft SQL Server's main-memory OLTP engine.

Hekaton. The Hekaton engine is integrated into SQL Server; it is not a separate database system. A user can declare a table to be memory-optimized which means that it will be stored in main memory and managed by Hekaton. A Hekaton table can have several indexes and two index types are available: hash indexes and ordered indexes (the Bw-tree). Hekaton is designed for high levels of concurrency but it does not rely on partitioning to achieve this; any thread can access any row in a table without acquiring latches or locks. All data structures in Hekaton are latch-free (lock-free) to avoid physical interference among threads.

Scenario. This demo involves running a large-scale enterprise OLTP workload on Hekaton. The workload involves updates and queries that exercise the Bw-tree. Figure 4 depicts the SQL Server Management Studio used to run the workload and provide timing results. We run two versions of the workload: (1) *Memory optimized* that defines all tables using the `MEMORY_OPTIMIZED` keyword, denoting they are managed by the Hekaton engine; (2) *Regular* that uses regular SQL Server tables with a buffer pool large enough to fit the OLTP workload in memory. The attendee will be able to witness the significant performance improvement that the Bw-tree and Hekaton exhibit compared to a traditional database architecture running completely in-memory.

3.2 Bw-Tree as a Key-Value Store

This demo scenario showcases the Bw-tree as a standalone key-value store. We provide a side-by-side comparison of the Bw-tree with two different ordered index architectures:

Figure 5: Demo scenario workload driver

BerkeleyDB B-tree. To show that the Bw-tree is an efficient flash-aware key-sequential index, we compare it to the BerkeleyDB B-tree known for its good performance as a standalone storage engine. We use the C implementation of BerkeleyDB running as a standalone B-tree index sitting over a buffer pool cache that reads and writes from disk at page granularity, representing a typical B-tree architecture. We run BerkeleyDB in non-transactional mode (meaning better performance) that supports a single writer and multiple readers using page-level latching (the smallest latch granularity in BerkeleyDB, and indeed in most database engines) to maximize concurrency.

Latch-free skip list. To show that the Bw-tree serves as a very efficient memory-only latch-free ordered index, we compare it to a latch-free main memory skip list implementation [11]. The skip list has become a popular key-sequential index choice for in-memory databases[2] since it can be implemented latch-free, exhibits fast insert performance, and maintains logarithmic search cost. Our implementation installs an element in the bottom level (a linked list) using a CAS to change the pointer of the preceding element. It decides to install an element at the next highest layer (the skip list towers or "express lanes") with a probability of $\frac{1}{2}$.

3.3 Benchmark Machine and Workloads

The demo uses workloads from two real-world applications from within Microsoft:

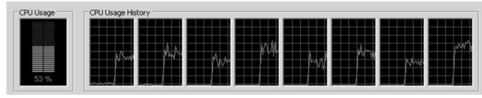
Xbox LIVE. This workload contains 27 Million `get-set` operations obtained from Microsoft's Xbox LIVE Primetime online multi-player game [13]. Keys are alpha-numeric strings averaging 94 bytes with payloads averaging 1200 bytes. The read-to-write ratio is approximately 7.5 to 1.

Enterprise storage deduplication trace. This workload comes from a real enterprise deduplication trace used to generate a sequence of chunk hashes for a root file directory and compute the number of deduplicated chunks and storage bytes. This trace contains 27 Million total chunks and 12 Million unique chunks, and has a read to write ration of 2.2 to 1.

When comparing the Bw-tree with the skiplist, we run the workloads in *memory-only mode* by removing the flash layer of the Bw-tree, forcing it to run completely in memory.

[2]For example, the MemSQL in-memory database uses a skip list as its key-sequential index [10]

(a) Bw-tree CPU monitor



(b) BerkeleyDB CPU monitor

Figure 6: CPU monitor.

When comparing the Bw-tree with BerkeleyDB, we run the workloads in *flash mode* by setting system memory limits low enough to force each system to spill pages to flash storage.

3.4 Demo Application

Our primary demo application is a workload driver responsible for playing our real-world workloads over each index (as depicted in Figure 5). The driver also reports the performance for each system.

In order for the demo attendee to witness the speed of each system, the demo application "flips" a tile after completion of every 1M operations. Upon workload termination the application reports the throughput achieved by each system. The demo also uses a live CPU monitor, depicted in Figure 6 so the demo attendee may see the CPU utilization achieved by each system. For example, the Bw-tree (and Skiplist) achieves 100% CPU utilization when the workloads run in *memory-only* mode, and close to a 100% utilization for the *flash mode* workload (Figure 6(a)). Meanwhile, BerkeleyDB achieves roughly 60% CPU utilization (Figure 6(b)) due to latching and and in-place updates both in memory and on flash. Our demonstration scenario shows the Bw-tree achieving up to a 3x and 19x speedup over the skiplist and BerkeleyDB B-tree, respectively.

4. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[2] D. Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[3] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *SIGMOD*, pages 25–36, 2011.

[4] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[5] Amazon DynamoDB. `http://aws.amazon.com/dynamodb/`.

[6] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*, pages 123–133, 2011.

[7] J. J. Levandoski, D. B. Lomet, and S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, 2013.

[8] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, pages 302–313, 2013.

[9] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling Transaction Services in the Cloud. In *CIDR*, pages 123–133, 2009.

[10] MemSQL Indexes. `http://developers.memsql.com/docs/1b/indexes.html`.

[11] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, 1990.

[12] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[13] Xbox LIVE Primetime. `http://www.xboxprimetime.com`.