The BW-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage

Justin Levandoski Sudipta Sengupta
Microsoft Research
Redmond, WA
{justinle,sudipta}@microsoft.com

Abstract

The Bw-Tree is a high performance latch-free B-tree index that exploits log-structured storage. Its design addresses two emerging hardware platform trends. (1) Multi-core and main memory hierarchy: the Bw-tree is completely latch-free; it performs state changes (e.g., record updates, splits) as "deltas" prepended to prior state, installing new state via an atomic compare-and-swap instruction on an indirection page address mapping table. This improves performance by avoiding thread latch blocking while also improving multi-core cache behavior. (2) Flash storage: the Bw-tree organizes storage in a log-structured manner (using "delta" records) on flash to exploit fast sequential writes and mitigate adverse performance impact of random writes. This also reduces write amplification and extends device lifetime. This article provides an overview of the Bw-tree architecture and its technical innovations that achieve very high performance on modern hardware.

1 Introduction

A B-tree index supports high performance key-sequential access to both individual keys and designated subranges of keys. It is the combination of random and range access that has made B-trees the indexing method of choice within database systems and stand-alone atomic record stores. But the hardware infrastructure for which the B-tree was designed has changed dramatically. That infrastructure used processors whose uni-processor performance increased with Moore's Law, limiting the need for high levels of concurrency on a single machine. It used disks for persistent storage. Disk latency is now analogous to a round trip to Pluto [14]. Those days are gone. In this article, we provide an overview of the Bw-tree [12], a new B-tree whose design enables very high performance in the new hardware environment that has recently emerged. The Bw-tree addresses two important trends:

Design for multi-core. We now live in a multi-core world where uni-core speed will at best increase modestly. We need to better exploit a large number of cores by addressing at least two important aspects:

1. Multi-core CPUs mandate high concurrency. But, as the level of concurrency increases, latches are more likely to block, limiting scalability [1].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

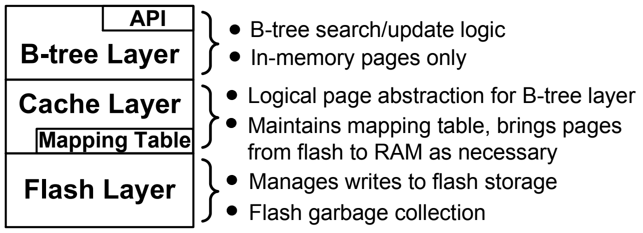

API
B-tree Layer
Cache Layer
Mapping Table
Flash Layer
B-tree search/update logic
In-memory pages only
Logical page abstraction for B-tree layer
Maintains mapping table, brings pages from flash to RAM as necessary
Manages writes to flash storage
Flash garbage collection

Figure 1: The Bw-tree architecture.

2. Good multi-core processor performance depends on high CPU cache hit ratios in a hierarchical cache architecture. Updating memory in place results in cache invalidations, so how and when updates are done needs great care.

Addressing the first issue, the Bw-tree is latch-free, ensuring a thread never yields or even re-directs its activity in the face of conflicts. Addressing the second issue, the Bw-tree performs "delta" updates that avoid updating a page in place, hence preserving previously cached lines of pages.

Design for Modern Storage Devices. Hard disk I/O operations per second (IOPS) performance is limited by disk seek time. Flash storage offers higher IOPS at lower cost. This is key to reducing costs for OLTP systems. Indeed, current storage systems such as Amazon's DynamoDB include explicit ability to exploit flash [5]. The Bw-tree targets flash storage as well. Flash has some performance idiosyncracies, however. While flash has fast random and sequential reads, it needs an erase cycle prior to re-write, hence uses "copy-on-write" and garbage collection mechanisms to provide the illusion of "in-place" updates. This makes random writes significantly less efficient than sequential writes in terms of both performance and wearing out the device [8]. While flash SSDs use a mapping layer (FTL) in order to hide the "copy-on-write" nature of writes and provide the abstraction of "logical" page, this cannot hide the noticeable slowdown in performance associated with random writes. Even high-end FusionIO drives exhibit a 3x faster sequential write performance than random writes [3]. The Bw-tree performs log structuring itself at its storage layer. This approach avoids dependence on the FTL and ensures that our write performance is high for both high-end and low-end flash devices.

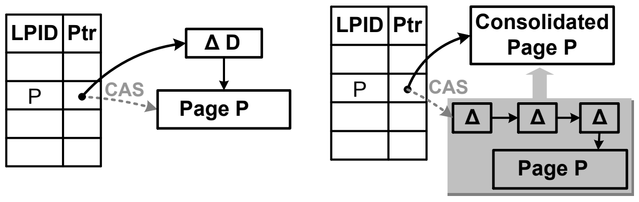
This article provides an overview of the Bw-tree. We first discuss the high-level architecture of the Bw-tree. We then describe the novel techniques used to achieve in-memory latch-free behavior. Next, we provide a brief discussion of our log-structuring technique to achieve persistence, and how the Bw-tree supports transactional functionality when part of a lager system (e.g., a deuteronomy-style architecture [11, 13]). We end by discussing the Bw-tree's role in Hekaton, Microsoft SQL Server's memory-optimized database engine.

2 The Bw-Tree Design

This section highlights the novel features of the Bw-tree. We begin by presenting the Bw-tree architecture and then discuss specific techniques that make the Bw-tree latch-free and optimized for log structured storage.

2.1 Bw-tree Architecture

The Bw-tree is a classic B+-tree [2] in many respects. It provides logarithmic access to keyed records from a one-dimensional key range, while providing linear time access to sub-ranges. Figure 1 depicts the Bw-tree architecture. The top *B-tree* layer provides the access method API, and is responsible for the search, record update, and structure modification logic common to a B-tree. The *cache layer* serves the *B-tree* layer with in-memory pages; it is responsible for swapping out pages into flash under memory pressure and bringing them back into memory when the B-tree layer accesses them. It is also responsible for installing "delta" updates on a page in a latch-free manner. The *flash layer* implements our log-structured store (LSS).



(a) Delta updates (b) Consolidating a page

Figure 2: Delta updates and consolidation.

This design is extremely versatile since (1) it is architecturally compatible with existing database kernels; (2) it is suitable as a stand-alone atomic record store, or "data component" in a decoupled transactional system [11, 13]; (3) due to its latch-freedom, it can serve as an efficient range index in a main-memory database (that is independent of the flash layer). In fact, the Bw-tree is the ordered index within Microsoft's Hekaton main-memory optimized database engine [7].

2.2 The Mapping Table

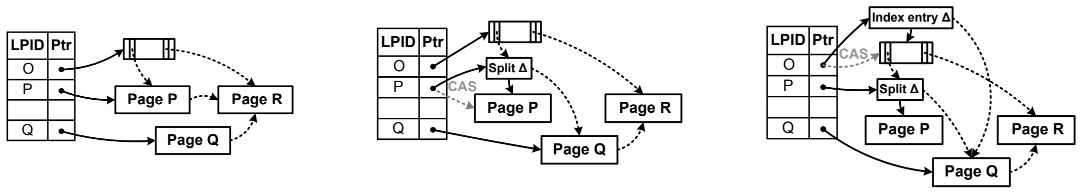
Our cache layer maintains a *mapping table*, that maps logical pages to physical pages; logical pages are identified by a logical "page identifier" or PID. The mapping table translates a PID into either (1) *a flash offset*, the address of a page on stable storage, or (2) *a memory pointer*, the address 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.

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.

2.3 Latch-Free In-Memory Operations

In our Bw-tree design, threads never block on Bw-tree pages in memory because we do not use latches. Being latch-free permits us to drive multi-core processors to close to 100% utilization. Instead of latches, we install state changes using compare and swap (CAS) instructions[1]. The Bw-tree provides an asynchronous operation completion pathway when it needs to fetch a page from stable storage (the LSS); this pathway would be taken rarely when the workload has a working set that is captured by the cache layer's page swapout mechanism (and fits within the provided memory limit).

This persistence of thread execution helps preserve core instruction caches, and avoids thread idle time and context switch costs. Further, the Bw-tree performs node updates via"delta updates" (attaching the update to an existing page), not via update-in-place (updating the existing page memory). Avoiding update-in-place reduces CPU cache invalidation, resulting in higher cache hit ratios. Reducing cache misses increases the instructions executed per cycle. This section describes our techniques for eliminating latches for in-memory operations.



(a) Creating sibling page Q (b) Installing split delta (c) Installing index entry delta

Figure 3: Latch-free split example.

2.3.1 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 installs 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 or flushing it to stable storage). 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". We garbage collect the prior page (with deltas) by placing it on a pending list to be reclaimed when safe (i.e., when no other threads may access it) using an epoch based mechanism. Garbage collection and the CAS failure protocol are described in our full paper [12].

2.3.2 Structure Modifications

Structure modification operations (SMOs) such as node splits and 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 installable via a CAS. We describe a latch-free page split. Page merge SMOs are described in [12].

The Bw-tree employs the B-link atomic split installation technique that works in two phases [10] as depicted in Figure 3. We split an existing page P by first creating a new page Q and initializing it with the records of the upper half of the key range (Figure 3(a)). We install Q in a new entry in the mapping table that is not yet visible to the rest of the tree. We then install a "split delta" on P (Figure 3(b)) that logically describes the split and provides a side-link to new sibling Q. We then post a (seach key, PID) index term for Q at parent O with a delta record, again using a CAS (Figure 3(c)). In order to make sure that no thread has to wait for an SMO to complete, a thread that sees a partial SMO will complete it before proceeding with its own operation.

[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*.

2.4 Log Structured Store

2.4.1 Caching

The cache layer 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 layer. 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 [11, 13], or to reduce memory usage. Flushing is also required for implementing structure modifications correctly in the face of crash and subsequent recovery. Flushing and "swap out" of a page installs a flash offset in the mapping table and permits reclaiming page memory.

2.4.2 Storage Management

Our LSS has the usual advantages of log structuring [15]. Pages are written sequentially in a large batch, eliminating any write bottleneck by reducing the number of separate write I/Os required. The cache manager marshals bytes from the pointer representation of the page in main memory into a linear representation that can be written to the flush buffer. An important difference from [15] is the notion of *incremental flushing* that allows only the changed portion of a page to be flushed, thus further increasing the efficiency of log-structured writes and reducing the amount of garbage created on flash, and hence, write amplification.

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 fit in the flush buffer, and hence reducing the number of I/O's per page. There is a penalty on reads, however, as a page read into memory requires multiple I/Os to fetch the non-contiguous parts of the page. This penalty is mitigated by the very high random read performance of flash.

Garbage collection. When the log usage exceeds configurable thresholds, the LSS cleaner reclaims garbage records in the log. It operates from the earliest written portion of the log, passing over orphaned records and copying valid records to the end of the log. Delta flushing reduces pressure on the LSS cleaner by reducing the amount of storage used per page. This reduces the "write amplification" that is a characteristic of log structuring. During cleaning, LSS makes pages and their deltas contiguous on flash for improved access performance.

2.5 Managing Transactional Logs

The Bw-tree is an atomic record store that can be "plugged" into a transactional system. When part of such a system, the Bw-tree layer must manage the transactional aspects imposed on it. To do this, we tag each update operation with a unique identifier that is typically the log sequence number (LSN) of the update on the transactional log (maintained elsewhere, e.g., in a transactional component [11]). LSNs are managed so as to support recovery idempotence, i.e., ensuring that operations are executed at most once, and to support transaction log management.

A major issue in a transactional system is enforcement of the write-ahead log protocol (WAL). A decoupled Deuteronomy [11] architecture enforces the WAL by a transactional component sending an LSN value, called the EOSL, representing its end of stable log to a data component (e.g., the Bw-tree). The data component is then allowed to make all operations stable with LSNs *less than or equal to* EOSL. Like conventional systems, the Bw-tree flushes pages lazily while honoring the WAL. Unconventionally, we do not block when performing a checkpoint that requires flushing a page whose updates are more recent than (have LSNs greater than) the EOSL used to enforce WAL [12]. Instead, because the recent updates are separate deltas from the rest of the page (a property we guarantee in deciding when to consolidate a page), we can remove the “recent” updates (not on the stable transactional log) from pages when flushing.

3 Hekaton: The Bw-tree in Action

Hekaton is Microsoft SQL Server’s memory-optimized database engine targeting OLTP workloads [4]. The engine is designed for high levels of concurrency and achieves great performance, demonstrating speedups of greater than an order of magnitude on real customer workloads [6]. A number of novel technical advancements help achieve such performance. For instance, Hekaton uses a new optimistic, multi-version concurrency control technique to avoid interference among transactions [9]. Also, the engine uses latch-free (lock-free) data structures in order to avoid physical interference among threads. Hekaton supports two access methods: hash indexes and ordered indexes. The Bw-tree serves as Hekaton’s ordered index.

The Hekaton team considered several alternatives for its ordered index. One alternative was a skip list, which also provides key-ordered access, logarithmic search overhead, and can be implemented to provide latch-free updates. In several head-to-head performance evaluations, the Bw-tree consistently outperformed skip lists by a factor of 3-4x. Clearly, it did not achieve this advantage by merely being latch-free. Rather, the Bw-tree improvement stemmed from much better processor cache hit ratios, attributable to our no update-in-place approach, and the fundamental cache efficiency of a B-tree based structure. All experimental numbers, including cache hit comparisons, are reported in [12].

4 Acknowledgements

This is joint work with David Lomet. We would like to also thank the Hekaton team and several other product groups at Microsoft for providing us valuable feedback that improved the design of the Bw-tree.

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] How Fast is Project Codenamed Hekaton? It’s Wicked Fast. **http://tinyurl.com/bzjync9**.

[7] Hekaton Breaks Through. **http://research.microsoft.com/en-us/news/features/hekaton-122012.aspx**.

[8] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR ’09, pages 10:1–10:9, 2009.

[9] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4):286–297, 2012.

[10] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *TODS*, 6(4):650–670, 1981.

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

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

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

[14] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB Journal*, 4(4):603–627, 1995.

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