Lecture #06: OLTP Indexes (Tree Data Structures)

15-721 Advanced Database Systems (Spring 2020)
https://15721.courses.cs.cmu.edu/spring2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

The original B+Tree [1] from 1970s was designed for efficient access of data stored on slow disks. But if we assume that a database fits entirely in memory (including its indexes), then we may need to consider alternative data structures that are specifically designed for this operating environment. Such data structures could provide more efficient multi-threaded access.

2 T-Trees

The **T-Tree** [3] was one of the first attempts for creating a data structure designed for in-memory databases. It is designed to reduce the size of the index in exchange for more computational overhead during operations. The main idea of a T-Tree is that instead of storing copies of keys in nodes (as in a B+Tree), T-Trees store pointers to the their original values. In order to perform a comparison between the search key and a key in the index, the DBMS must follow the pointer to the tuple to retrieve the key. The overall architecture is similar to an AVL-Tree where threads perform breath-first search ordering of keys.

The T-Tree was proposed in 1986 from database researchers at University of Wisconsin–Madison. It is also used in **TimesTen** (originally called **Smallbase** [2]) and other early in-memory DBMSs developed in the 1990s. Although T-Trees are still used in some DBMSs designed for operating environments with limited memory (e.g., embedded devices), they are not commonly used in large-scale in-memory DBMSs.

Advantages:

- Uses less memory because it does not store keys inside of each node.
- Inner nodes contain key/value pairs (like B-Tree), which means the DBMS does not need to always traverse to the leaf nodes to find the matching key.

Disadvantages:

- Difficult to re-balance because keys can move either up or down the tree.
- Difficult to implement safe concurrent access.
- Not cache-friendly because threads chase pointers when scanning range or performing binary search inside of a node.

3 Bw -Tree

The Bw-Tree is a latch-free ("lock-free") indexing data structure designed by Microsoft Research for the Hekaton project [4]. In latch-free data structures, threads use atomic *compare-and-swap* (CAS) instructions instead of latches to access or modify critical sections.

It is not possible to make a latch-free B+Tree with sibling pointers. This is because threads may need to update sibling pointers during split and merge operations, and it is not possible to use CAS to atomically update multiple addresses. This is the problem that the Bw-Tree solves through an indirection layer.

Bw-Tree also uses an indirection layer, called the **Mapping Table**, to map (logical) page IDs to their physical address locations in memory. This indirection layer allows for CAS of physical locations of pages. Threads check the Mapping Table to find out where they need to go when traversing the tree in memory. If a thread wants to change the location of a page, it can just perform a CAS into a single memory address in the Mapping Table, and that updates all the pointers.

The Bw-Tree uses deltas to record changes made to single nodes known as **Delta Records**. It is similar to a B+Tree except it has two key differences. The first is that the tree does not allow for in-place updates. Instead, each update to a page produces a new delta that physically points to the *base page*, which acts as the head of the delta chain. The DBMS then installs deltas in physical address slot of Mapping Table using CAS.

Operations

We now describe the two basic operations for accessing and modifying the Bw-Tree.

Search:

- Traverse tree like a B+tree, perform comparisons along nodes.
- If Mapping Table points to delta chain, stop at first occurrence of search key.
- Otherwise, perform binary search on base page.

Delta Update:

- Since in-place operations are not allowed, each update to a new page produces a new delta.
- Delta physically points to the base page and other deltas.
- Install delta address in physical address slot of Mapping Table using CAS.
- If multiple threads try to install updates to the same page, then only one thread will succeed in installing their change. All other threads must retry their operation.

Garbage Collection

As threads modify the index and append new delta records to nodes, the node's will grow in length and make searches take longer. Thus, the DBMS needs to periodically compact these chains.

The first step is through *cooperative consolidation* where threads recognize that a chain is too long during a normal traversal and they compact it

Consolidation:

1. The thread a copy of the target page and then applies the deltas in reverse order to the new page.
2. The thread then updates the Mapping Table to have the node id point to memory address of the new page that it created. Using CAS ensures that the thread does not miss any new deltas that were added after the consolidation step started.
3. Lastly, the thread registers the old page and its delta chain as reclaimable with the tree's garbage collector.

After consolidation, the garbage collector needs to recycle old deltas that have been already applied and old pages. The Bw-Tree uses an *epoch-based garbage collection* scheme. This approach is also called *RCU* in Linux and widely used for its internal data structures.

- All operations are tagged with an **epoch**, which is a logical counter that keeps increasing.
- Each epoch tracks the threads that are part of it and the objects that can be reclaimed.

- A thread performing an operation joins an epoch prior to each operation and posts objects that can be reclaimed for the current epoch (not necessarily the one it joined).
- Garbage for an epoch is reclaimed only when all threads have exited the epoch.

Structure Modifications

Since the Bw-Tree is a self-balancing tree, so it needs to perform splits and merges. There are two additional delta record types to keep track of these changes.

Split Delta Record: Keeps track of where certain ranges of a key or a page can be found. Marks that a subset of the base page's key range is now located at another page and uses a logical pointer to that new page.

Separator Delta Record: Shortcut mechanism for higher parts of the tree. Provides information in the modified page's parent on what ranges to find the new page. This reduces wasted time where threads traverse the delta chain only to find out that the target key is located in a different node.

CMU OpenBw-Tree

The original Bw-Tree paper from Microsoft is missing important details on how to actually implement the data structure. Thus, CMU set out to implement its own version of the Bw-Tree[5] for the **Peloton** DBMS project. It includes some additional optimizations:

- **Pre-allocated Delta Records:** Use extra space in each node to store delta records. When there are no more available slots to store new deltas in a node, this triggers a consolidation on that node. This avoids the need for the DBMS to allocate memory for many small objects and avoids running into random locations in memory that may not be in the CPUs caches.
- **Mapping Table Extension:** The Mapping Table is not implemented as a dynamic hash table, but as a flat array as it is the fastest associative data structure. Allocating the full array for each index is wasteful; instead, we can use virtual memory to allocate the entire array without backing it with physical memory. Even if the entire array is allocated in the virtual memory, it is not allocated in the physical memory unless the entry of the array has been accessed. OS only allocates physical memory when threads access high offsets in the array.

References

[1] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121137, June 1979. ISSN 0360-0300. doi: 10.1145/356770.356776. URL https://doi.org/10.1145/356770.356776.

[2] M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson. Smallbase: A main-memory dbms for high-performance applications. Technical report, Hewlett-Packard Laboratories, 1995.

[3] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th international conference on Very large data bases*, pages 294–303, August 1986. URL http://www.vldb.org/conf/1986/P294.PDF.

[4] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE '13 Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, pages 302–313, April 2013. doi: https://dl.acm.org/citation.cfm?id=2510649.2511251.

[5] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 473–488, 2018. doi: 10.1145/3183713.3196895.