Lecture #09: OLTP Indexes – Part 2

15-721 Advanced Database Systems (Spring 2018)
http://15721.courses.cs.cmu.edu/spring2018/
Carnegie Mellon University
Prof. Andy Pavlo

1 Index Implementation Issues

Beyond just what type of data structure one uses for an in-memory database index, there are some additional design issues that one must deal with in order to use it in a real DBMS.

Memory Pools

We do not want threads to be calling `malloc` and `free` anytime we need to add or delete a node in our index.

If all the nodes are the same size (or a small number of fixed sizes), then the index can maintain a pool of available nodes.

- **Insert:** Grab a free node, otherwise create a new one.
- **Delete:** Add the node back to the free pool.

We need some policy to decide when to retract the pool size. This is non-trivial.

Garbage Collection

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

Approach #1 – Reference Counting:

- Maintain a counter for each node to keep track of the number of threads that are accessing it.
  - Increment the counter before accessing
  - Decrement it when finished
  - A node is only safe to delete when the count is zero
- This has bad performance for multi-core CPUs, as changing the counters causes a lot of cache coherence traffic.

Approach #2 – Epoch Garbage Collection:

- Maintain a global epoch counter that is periodically updated (e.g., every 10 ms).
- Keep track of what threads enter the index during an epoch and when they leave.
- Mark the current epoch of a node when it is marked for deletion.
- The node can be reclaimed once all threads have left that epoch (and all preceding epochs).
- Also known as Read-Copy-Update (RCU) in Linux.

Non-Unique Indexes

Every index needs to support non-unique indexes [1].

Approach #1 – Duplicate Keys:

- Use the same node layout as a unique index but store duplicate keys.

Approach #2 – Value Lists:

- Store each key only once and maintain a linked list of unique values.

Variable Length Keys

Not all keys will be the same length.

Approach #1 – External Pointers:

- Store the keys as pointers to the tuples attributes in the table heap.

Approach #2 – Variable Length Nodes:

- The size of each node in the index can vary.
- Requires careful memory management.

Approach #3 – Padding:

- Always pad the key to the max length of the key type.
- This wastes space and requires that all keys are the same length of the largest possible key in the index.

Approach #4 –Key Map / Indirection:

- Embed an array of pointer that map to the key + value list within the node.
- Think of this as the same as a slotted page in a disk-oriented DBMS.

Prefix Compression

Since keys are sorted in lexicographical order, there will be a lot duplicated prefixes. Store a minimum prefix that is needed to correctly route probes into the index.

2 Adaptive Radix Tree

The Adaptive Radix Tree (ART) was developed for the HyPer DBMS [2]. It uses digital representation of keys to examine prefixes one-by-one instead of comparing entire keys. The structure of every radix tree is deterministic (i.e., the layout of nodes will be the same regardless of the order that threads insert keys).

The ART index supports four different internal node types with different capacities. It packs in multiple digits into a single node to improve cache locality.

Key Ideas:

- The height of the tree depends on the length of the keys.
- Does not require re-balancing.
- The path to a leaf node represents the key of the leaf.
- Keys are stored implicitly and can be reconstructed from path.

Binary Comparable Keys

Not all attribute types can be decomposed into binary comparable digits for a radix tree. Thus, the ART index has to transform the keys into a binary comparable form.

- **Unsigned Integers:** Byte order must be flipped for little endian machines.
- **Signed Integers:** Flip two's complement so that negative numbers are smaller than positive.
- **Floats:** Classify into group (neg vs. pos, normalized vs denormalized), then store as unsigned integer.
- **Compound Keys**: Transform each attribute separately and then concatenate them together into a single byte-array.

3 Concurrent ART

HyPer's ART is **not** latch free [3]. There are two ways to make a concurrent ART that is thread-safe.

Approach #1 – Optimistic Lock Coupling (OLC):

- Optimistic crabbing scheme where writers are not blocked on readers.
- Writers increment a node's version counter when they acquire latch on that node.
- Readers can proceed if a node's latch is available. Before moving to the next node, the reader checks whether the latch's version counter has changed from when it checked the latch.

Approach #2 – Read-Optimized Write Exclusion (ROWEX):

- Each node includes an exclusive lock that blocks only other writers and not readers.
- Readers proceed without checking versions or locks.
- Every writer must ensure that reads are always consistent.
- Requires fundamental changes to how threads make modifications to index to make the scheme work. The changes will vary per data structure.

References

[1] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011. URL `https://www.nowpublishers.com/article/Details/DBS-028`.

[2] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE '13 Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, pages 38–49, April 2013. doi: https://dl.acm.org/citation.cfm?id=2511193.

[3] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *DaMoN '16 Proceedings of the 12th International Workshop on Data Management on New Hardware*, number 3, June 2016. doi: https://dl.acm.org/citation.cfm?id=2933352.