Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes

Maksym Semikin, maksym.semikin@tudeng.ut.ee

Institute of Computer Science, University of Tartu

November 23, 2017

Abstract

This essay is a review of a similarly titled article suggesting a composite indexing data structure [1]. Knowledge about the asymmetry of queries for database indexes allows to separately optimize for read-only and read-write scenarios to more efficiently use DRAM. Specific guidelines can be applied to a variety of data structures to build hybrid indexing architectures with different characteristics. Experiments confirm index memory size reduction by up to 70% without significant loss in performance.

Introduction

Online transaction processing database management systems (OLTP DBMS) and other in-memory stores require indexing structures for fast access to data. Indexes are most efficient when placed entirely in main memory. DRAM is expensive and infrastructures will benefit from reduced index sizes. However, indexes must remain fast, which excludes the use of compression. Authors suggest combining 2 data structures, one aimed at maximizing performance for most often queried data, second to compactly store less frequently used parts of the index. Standard indexing data structures can be used for dynamic stage, but since the static stage is read-only, we can further optimize its size and performance.

1 Hybrid index structure

There is an asymmetry that recently inserted or updated data is accessed and written more often than "cold" data. We can exploit this by designing the index structure to keep the most often queried subset of the index in fast dynamic structure while optimizing the rest of the data for read-only access in a static one. To keep the whole system efficient, periodic merges from dynamic to static stages are necessary (see Figure 1).

1.1 Key operations

Indexes support point and range read queries. In a hybrid approach, we first look up the value in dynamic stage and if not found there, we query the static part. Inserted values go directly to the dynamic stage. On update, if the original value is in the dynamic stage, it is updated in place. If the original value is in the static stage, the updated version is added to the dynamic part and will be found first when queried for. The redundant old value is kept in the static stage until the next merge. For the deletes, values in the dynamic stage are directly removed, while values in the static stage are marked to be removed and disposed on the next merge. We reduce the overhead of checking both stages for the presence of the value with a Bloom Filter which tells us whether we need to consult the static stage immediately.

2 Dual-Stage Transformation

Any existing indexing structure can be converted into a dual-stage combined index by following the steps [1]:

1. Select an order-preserving index structure (X) that supports dynamic operations efficiently for the dynamic stage.
2. Design a compact, read-optimized version of X for the static stage.
3. Provide a merge routine that can efficiently migrate entries from X to compact X.
4. Place X and compact X in the dual-stage architecture as shown in Figure 1.

3 Designing static stage

Having the property that static stage is modified only periodically, we can further optimize its memory footprint. We are interested in data structures that support fast lookups, are memory-efficient and provide a way to be efficiently merged with the dynamic part of the index. A procedure for building the static data structure from a conventional one includes:

1. Compaction - Remove duplicated entries and make every allocated memory block 100% full.
2. Structural Reduction - Remove pointers and structures that are unnecessary for efficient read-only operations.
3. Compression - Compress parts of the data structure using a general purpose compression algorithm.

4 Merge

The final piece of the Dual-Stage structure is the merge procedure. Authors present a fully-blocking merge algorithm which scales linearly in benchmarks. Input for the merge is the part of the dynamic structure and the whole static structure. Both of them are sorted. One challenge of the in-memory merge is to fit in memory. For this, only an array of size of the merged data from the dynamic data structure is allocated. Then, an in-place merge sort of 2 arrays is performed.

4.1 Merge design choices

We can either merge all entries from the dynamic stage or only cold ones. Merge-Cold keeps a "cache" of hot values accessible, but merges are more complicated and more frequent, while Merge-All is a universal solution and is used later in the evaluation. Another decision is when to trigger merges. The first option is to trigger them after the ratio of dynamic to static stage sizes reaches a threshold. Second is to trigger merge when the size of the dynamic stage reaches a fixed number. The first approach is preferable, as it generalizes better and leads to fewer merges.

5 Application to existing indexing data structures

Dynamic versions of B+Tree, Masstree [2], Skip list [3] and Adaptive Radix Tree (ART) [4] were adapted for static use with the guidelines presented above. Duplicated data and extra-allocated space were removed. Data structures were updated for a denser representation. Where possible, pointers were replaced with calculated positions from the contiguous alignment of nodes in memory. Compression was applied to leaf nodes, but appeared to hinder the performance and was mostly ignored in the evaluation.

6 Evaluation

First, compacted data structures were tested with The Yahoo! Cloud Serving microbenchmarks [5] for read performance and storage overhead as compared to original data structures. Microbenchmarks use 64-bit random integers, 64-bit monotonically incremented integers and email records as keys and 64-bit integers as values. As a result, compaction reduced the memory footprint by up to 71% as well as improved performance. Aligned data structures made better use of caching.

Then, insert-only, read-write, read-only and scan-insert benchmarking scenarios were used to compare hybrid indexes to original data structures. Results showed that all hybrid indexes achieved comparable throughputs to their original structures while consuming 30% to 70% less memory.

Finally, the hybrid B+tree index was evaluated in the context of H-Store OLTP DBMS [6]. The goal is to compare the hybrid index to the original B+tree index used in H-Store. The experiment used built-in benchmarks of H-Store and considered both the case when data fit into DRAM and when it overflows it. When faced with insufficient memory, H-Store uses "anti-caching" to move cold database tuples to disk. However, H-Store has no solution for when the index itself does not fit into memory.

With database fitting into available DRAM, the hybrid index occupied 40% to 65% less space than B+tree and caused only 1-10% performance drop. With workloads larger than memory and enabled "anti-caching", hybrid index showed higher throughput of transactions per time period (up to twice as much). As index itself occupied less memory, the point where "anti-caching" became needed occurred later than when using the plain B+tree. Even after this point, the throughput of H-Store with hybrid index remained higher due to more tuples accessible from main memory.

Conclusion

The dual-stage architecture allows optimizing both dynamic and static use of database indexes to preserve memory and thus fit more data in DRAM. The optimization is possible due to "hot" data being manipulated more often than "cold" data in the typical usage of OLTP systems. Dual-Stage Transformation is a set of steps to convert any indexing data structure to a hybrid. Future research could explore using Succinct data structures to further reduce the DRAM usage. Also, non-blocking merging techniques could be introduced.

Acknowledgement

The work is supported by the Dora Plus Estonian government programme.

References


