Learning Augmented Index Structure Design

Fuheng Zhao
fuheng_zhao@ucsb.edu
UC Santa Barbara

ACM Reference Format:
Fuheng Zhao. 2018. Learning Augmented Index Structure Design. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

Bio. I'm a PhD student at UC Santa Barbara studying data systems, with internship experiences at Snowflake and Gray System Lab [1]. Currently, I'm working on translating natural language questions to SQL queries using large language model. I would love to attend the International Workshop on High Performance Transaction Systems this summer to share ideas and network with peers in the field. In this proposal submission, I would like to share my thoughts on how learning based algorithms can be potentially used to augment the index structure of a key-value storage engine.

Introduction. Key-value (KV) stores are commonly used as the storage layer in modern large-scale data systems. Variations of B tree [1] and Log Structured Merge Tree (LSM tree) [8] are the two go-to index structures that power today's on-disk KV stores [3–6].

B trees have the property of in-place update and are known to be read-optimized. When internal nodes are cached, reading a target key only requires one disk I/O. Variations of the B tree, such as the B+ tree, keep all keys sorted at the leafs, and hence, retrieving keys from a range can be efficiently performed. The worst-case write amplification for the B tree is $O(B)$ (where B is the block size), as a single insert may cause an entire leaf block to be rewritten. LSM-tree offers high write throughput by applying out-of-place updates. It organizes data into levels of logs with exponentially larger capacity, and the last level contains a significant portion of the total data stored. Since most keys need to be written L times, from level 1 to level L, the write amplification for the LSM tree scales linearly with L. Since key boundaries may overlap across levels, $O(L)$ (L denotes the number of levels) need to be examined to find a target key.

Learning Augmentation. Let's assume there is a learned oracle (e.g., a neural network) based on past workload traffic that can emit the key distribution of reads, D_R, and the key distribution of writes, D_W. Let's also assume these distributions are highly accurate and static. In this case, a straightforward index structure design is to construct a B tree on the keys that are likely to be read (i.e., $P_{D_R}(.)$ > 0) and keep all other keys in a long append-only log. Under these assumptions, the average read cost is $O(1)$ and the average write cost is $O(B \cdot (\sum_x P_{D_W}(x) | P_{D_R}(x) > 0))$.

[1]Personal website: zhaofuheng.github.io

However, learned oracles often give noisy outputs. Since predictions are never perfect, storing data that are unlikely to be read into a long append-only log can cause the entire log to be examined when such data needs to be read. As a result, storing these data in an LSM tree-based storage engine such as Rocks DB is a better choice.

Moreover, since distributions may dynamically change, the data interchange between the B tree and the LSM tree needs to occur. Interestingly, on-fly transitions between the B tree and the LSM tree have been thoroughly studied in [7], in which they considered transforming an entire LSM tree into a B tree and vice versa. The proposed methods can also work for a particular key range.

Data Transition. When the oracle is updated, the prediction results may change, and data must be transited. At a high level, from the LSM tree to the B tree (when a key range becomes more likely to be accessed), one can either force a full compaction of the key range in the LSM tree and then insert these compacted key ranges into B tree, or read the keys in the range from the largest level in LSM tree and then perform L batch inserts into the B tree. From the B tree to the LSM tree (when a key range becomes unlikely to be accessed) can be done by converting the keys in the leaf levels of a B-Tree into a single run in the LSM tree.

Key Selections. Let α denote the percentage of keys that can be retrieved from the B tree with $O(1)$ disk I/O, and hence the read amplification on average is $O(\alpha + (1-\alpha)L)$ where $\alpha \in [0,1]$. We can formulate the construction of the B tree (i.e., what keys should be stored in the B tree) as an optimization problem, as shown in Equation 1, in which the objective is to minimize the write cost while maintaining the desired read performance.

$$\begin{aligned} &minimize \sum_x (P_{D_W}(x) | x \in Btree) \\ &s.t. \sum_x (P_{D_R}(x) | x \in Btree) \geq \alpha \end{aligned} \tag{1}$$

Workflow. When an oracle is trained based on past traffic [2], it can be used to determine the key boundaries of the B and LSM trees based on Equation 1. The key boundaries of the B tree can perhaps be stored in memory using a tire, especially when there are localities in the key ranges. In addition, for key ranges that have different classifications between the newly trained oracle and the previous oracle, the transitions will occur on the fly.

Discussion. In this proposal, we discuss the opportunity of how learning can be helpful in index design, assuming there exists an oracle that can produce accurate key distributions. Based on the oracle, one can construct a B tree paired with a LSM tree such that the design minimizes the write amplification while achieving the desired read performance. In addition, one can further consider the point read and the range read marginal distributions to leverage hash based index [2] in the design.

[2]Trained in windowed fashion to capture the change in distributions.

REFERENCES

[1] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 107–141.

[2] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

[3] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[4] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. 2010. InnoDB database forensics. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 1028–1036.

[5] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3535–3547.

[6] Google. 2011. *levelDB.* https://github.com/google/leveldb

[7] Varun Jain, James Lennon, and Harshita Gupta. 2019. Lsm-trees and b-trees: The best of both worlds. In *Proceedings of the 2019 International Conference on Management of Data*. 1829–1831.

[8] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.