Learned Index: A Comprehensive Experimental Evaluation (Technical Report)

In the technical report, we add experimental results that are not included in the main text due to space limitations.

1 CONCURRENCY SCENARIO EVALUATION

In this section, we report the performance under various insert factors. And we show the lookup (on Face/Osmc/Amzn datasets) and insert throughout (on Amzn dataset) after skewed inserts.

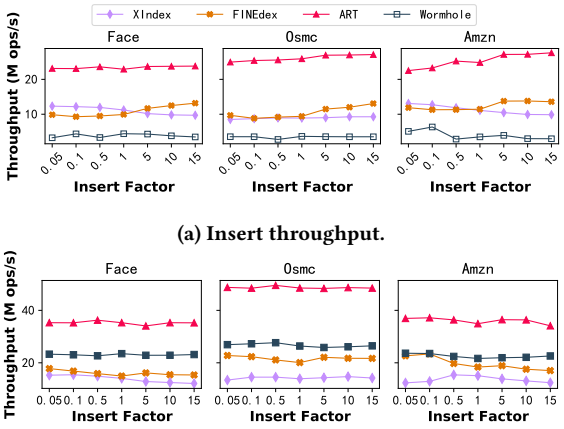
1.1 Performance Under Various Insert Factors

Figure 1 shows the insert and lookup throughput by varying the insert factor. We have three observations. First, learned indexes cannot outperform traditional indexes both in inserts and lookups, which is similar to the experimental results in our paper. Second, we find that XIndex performs higher insert throughput than FINEdex for low insert factors, while FINEdex performs higher for high insert factors. There are two reasons. (i) More structural modifications are triggered when the insert factor is high. FINEdex adopts more fine-grained optimizations for both intra-node (e.g., pair-level buffer) and inter-node concurrency (e.g., buffer-train-merge algorithm), which reduce thread collision and improve insert throughput. (ii) When the insert factor is low, FINEdex allocates more buffers than XIndex, which reduces the insert throughput. Third, FINEdex performs higher lookup throughput than XIndex after inserts. There are two reasons. (i) FINEdex adopts smaller buffers than XIndex, which reduces the search overhead within buffers. (ii) When XIndex splits the node into two new nodes during structural modifications, it links the new nodes. So XIndex has to scan the node list during lookups and performs lower lookup throughput.

1.2 Performance Under Skewed Insert

For the *hotspot* mode, the inserted pairs are evenly and randomly sampled from a small part of the dataset D. Specifically, if we sort D by the keys in an ascending list, the inserted keys are sampled between two array indexes l, r of the list. We define $hotspot\ \ ratio = \frac{r-l}{|D|}$, where $|D|$ is the number of pairs in D. Smaller *hotspot ratio* indicates more skewed inserted data. Since we insert 10M keys and the dataset includes 200M keys, we set the hotspot ratio no smaller than 5%. Besides, we evaluate the indexes on different hotspot ratios, so as to obtain a line chart with a smooth slope like Figure 2 (b).

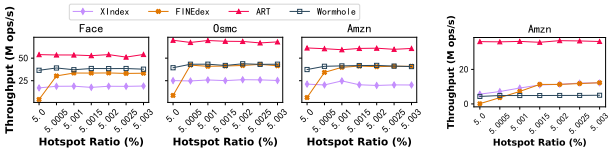
Figure 2 shows the insert and lookup throughput under different hotspot ratios. We have two observations. First, learned indexes cannot outperform traditional indexes, which is similar to the experimental results in our paper. Second, we find that XIndex performs more stable lookup throughput than FINEdex as the hotspot ratio decreases. That is because the structural modification of XIndex does not change the index height (always 2). While during structural modification, FINEdex may replace the full buffer with a new node and the index grows higher.



(a) Insert throughput.

(b) Lookup throughput after inserts.

Figure 1: Concurrent performance varying with insert factor.



(a) Lookup throughput after inserts.

(b) Insert throughput.

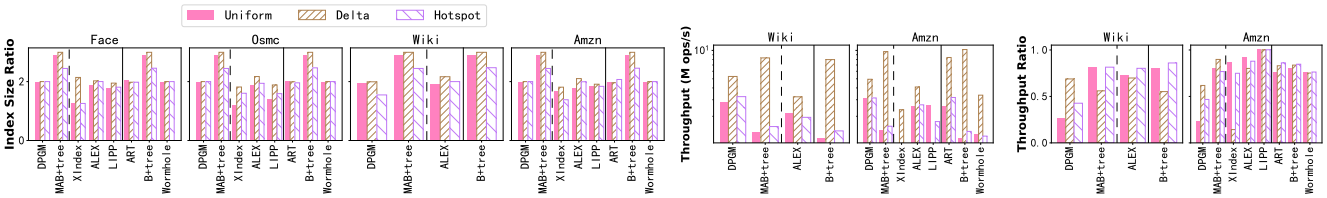
Figure 2: Concurrent performance varying with hotspot ratio.

2 DYNAMIC SCENARIO EVALUATION

In this section, we first demonstrate the performance (on Wiki/Amzn datasets) and index size changes under different insert patterns (on Face/Osmc/Wiki/Amzn datasets). Next, we analyze the micro-architectural behaviors of hybrid lookup/range/insert operations on Osmc dataset.

2.1 Index Performance And Index Size

Figure 3 shows the performance and index size under three insert patterns, i.e., uniform, delta, hotspot. We evaluate the insert throughput on Wiki/Amzn datasets in Figure 3 (b), and the results are similar to those in our paper. Next, we evaluate the lookup throughput (in Figure 3 (c)) and index size change before/after inserts (in Figure 3 (a)). We have three observations. First, the lookup throughput ratios of most indexes are larger than 0.5, and those index size ratios are not larger than 2 (insert factor is 1.0). This reflects that they trigger structural modifications in a proper frequency, which improves the lookup throughput without reducing the space efficiency. Second, indexes using fullness-based modifications can be less robust than others. For example, for uniform pattern, the lookup throughput of DPGM decreases most on both



(a) Index size ratio of after insert to before insert.

(b) Insert throughput.

(c) Lookup throughput ratio of after insert to before insert.

Figure 3: Index size and performance of different insert patterns. The solid line divides into learned (left) and traditional indexes (right). The dotted line divides learned indexes by structural modification, i.e., fullness-based (left), and others (right).

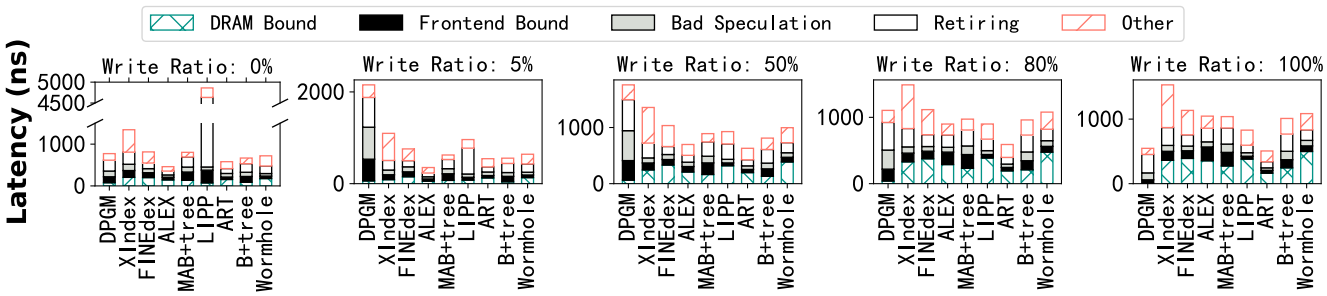


Figure 4: Average latency of different workloads (broken down by micro-architectural metrics) on Osmc dataset.

datasets. That is because whenever the inserted keys accumulate to a particular number, DPGM will rebuild a PGM index. Since the inserted keys of uniform pattern are randomly distributed in a large range, the linear models of the PGM index cannot fit well the key-to-position mapping and DPGM requires a higher height, which slows down the lookups. Third, We find that the index size of B+tree and MAB+tree increases greatly after inserts. Because during the bulk loading, they first initialize a node with the first pair and then add the next pairs into it until that node becomes full. They then store the next pair in the next node, and iteratively process it. As a result, they have a lot of full nodes after bulk loading, so that node splits are frequently triggered during inserts and the index size increases significantly.

2.2 Micro-architectural Analysis

The micro-architectural behaviors of indexes are similar on Wiki/Amzn datasets to those on Face, and we show the latency break-down on Osmc dataset in Figure 4. We have three observations.

First, LIPP performs very large retiring in read only and read heavy workloads. That is because whenever more than two keys are mapped to a position, LIPP will create a new node to store them. Since Osmc has very complicated distribution, the frequent key conflicts force LIPP to scatter the keys in a large number of nodes. During the scan of range query, LIPP has to access many nodes, which greatly increases instructions (retiring). Besides, ALEX performs lowest latency. There are two reasons. (*i*) ALEX searches only at leaf nodes, which reduces brach-instruction prediction (bad speculation) and instruction re-fetching between model prediction and position search (frontend bound). (*ii*) ALEX links the leaf nodes in a sorted sequence to facilitate range query, which decreases instructions (retiring) for index node access.

Second, apart from LIPP, we find that core bound (14.3% in Osmc, 7.3% in Face) and bad speculation (13.3% in Osmc, 9.5% in Face) occupy relatively large parts of the latency for read-only workloads, and are much larger than those in Face. We explain the core bound and bad speculation respectively. (*i*) The complicated distribution of Osmc causes more prediction errors for learned indexes, and thus more position searches. During searches, the index iteratively reduces the search range, and each iteration depends on the execution results of the previous iteration, where a lot of data dependencies occur among instructions (core bound). (*ii*) Since Osmc data is complicated, some learned indexes increase their height to better fit the key-to-position mapping. As a result, they conduct more model predictions and position searches, which make branch-instruction prediction more difficult (bad speculation).

Third, for balanced, write heavy and write only workloads, the micro-architectural behaviors are similar to those in Face dataset. That is because the insert performance mainly depends on the insert strategy and structural modification algorithm, and the dataset distribution has smaller effects. Similarly, we find that the traditional index ART performs lowest latency in balanced, write heavy and write only workloads. There are two reasons. (*i*) Compared with ALEX, ART involves fewer nodes (no more than 3 in most cases) and does not require to retrain the data fitting models during structural modifications, which decrease instructions (retiring) and cache misses (DRAM bound). (*ii*) DPGM adopts logarithmic merge method for structural modifications, which changes many PGM components and requires a lot of instructions (retiring).