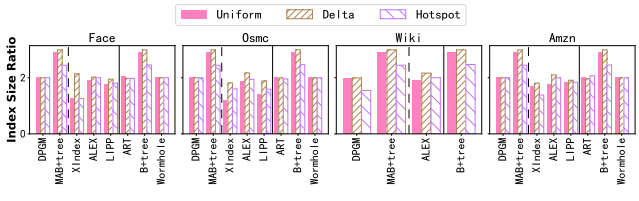
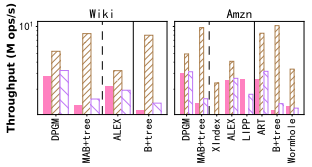
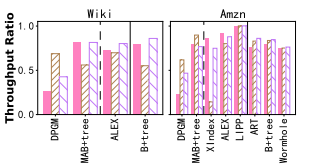
Learned Index: A Comprehensive Experimental Evaluation (Technical Report)



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



(b) Insert throughput.



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

Figure 1: Performance and index size of different insert patterns. The solid line divides into learned indexes (left) and traditional indexes (right). The dotted line divides learned indexes by structural modification, i.e., fullness-based (left), and others (right).

1 DYNAMIC SCENARIO EVALUATION

In this section, we first demonstrate (*i*) the index performance and (*ii*) the index size changes when conducting insert operations on `Wiki` and `Amzn` datasets. Next, we further analyze the micro-architectural behaviors of hybrid lookup/range/insert operations on datasets that have not been well discussed in paper.

1.1 Index Performance And Sizes

Figure 1 shows the throughput and index size under three insert patterns, i.e., uniform, delta, hotspot. We evaluate the insert throughput in Figure 1 (b), and the results are similar to those in our paper. Next, we evaluate the lookup throughput and index size change before/after inserts with 1.0 insert factor in Figure 1 (c) and Figure 1 (a). We have the following observations.

Index Performance. First, the lookup throughput ratios of most indexes are larger than 0.5, and those index size ratios are not larger than 2. 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 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 require a higher height, which slows down the lookups.

Index Size Changes. We find that the index size of B+tree and MAB+tree increases greatly after inserts. That is 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. As a result, they have a lot of full nodes after bulk loading, so that many node splits are frequently triggered during inserts and the index size increases significantly.

1.2 Micro-architectural Analysis

Since the micro-architectural behaviors of indexes are similar on `Wiki`/`Amzn` datasets to those on `Face`. we show the latency breakdown on `Osmc` dataset inFigure 2. 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 common 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 the instruction count (retiring).

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 reduce 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 makes 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.

2 CONCURRENCY SCENARIO EVALUATION

In this section, we first report the experimental results under various insert factors. Next, we show the lookup throughput after skewed inserts, and the insert/lookup throughput on `Face`/`Osmc`/`Amzn` datasets.

2.1 Performance Under Various Insert Factors

Figure 3 shows the insert and lookup throughput by varying the insert factor. We have two observations. First, learned indexes cannot

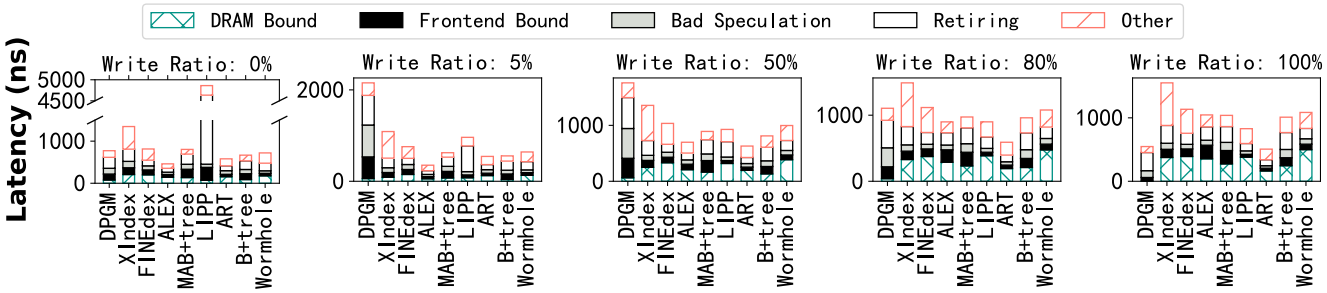
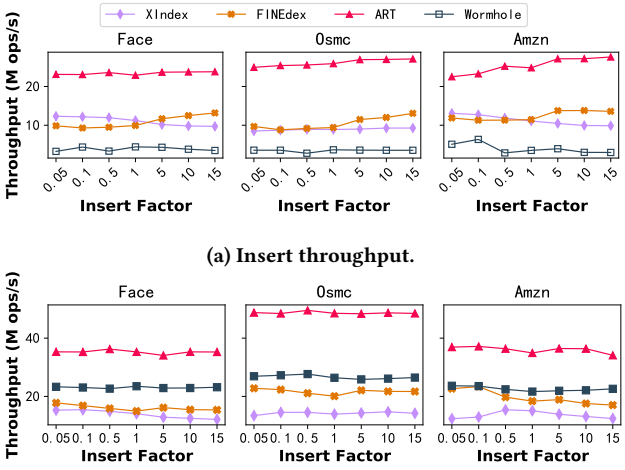


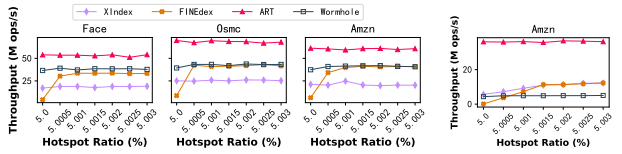
Figure 2: Average latency of different workloads (broken down by micro-architectural metrics) on `Osmc` dataset.



(a) Insert throughput.

(b) Lookup throughput after inserts.

Figure 3: Insert and lookup throughput varying with insert factor.



(a) Lookup throughput after inserts. **(b) Insert throughput.**

Figure 4: Insert and lookup throughput varying with hotspot ratio.

outperform traditional indexes both in inserts and lookups, which is similar to the experimental results in our paper. Second, we find that XIndex performs better than FINEdex for low insert factors, while FINEdex performs better for high insert factors. There are two reasons. (*i*) More structural modifications are triggered when the insert factor is high, and FINEdex adopts more fine-grained optimizations for both intra-node and inter-node concurrency to reduce structural modification overhead. (*ii*) When the insert factor is low, FINEdex's overhead of allocating new pair-level buffers is more dominant, which reduces the insert throughput.

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. 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 smooth slope like Figure 4.

Insert/Lookup Throughput. Figure 4 shows the insert and lookup throughput under different hotspot ratios (smaller hotspot ratio indicates more skewed inserted data). 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 the structural modification of FINEdex may replace the full buffer with a new node and the index grows higher.