DILI: A Distribution-Driven Learned Index (Appendix)

A APPENDIX

Due to space limit, we present some minute details and relatively less important experimental results and analyses in this appendix.

A.1 Index Scalability on Read-Only Workloadss

To further investigate how different methods perform in terms of scalability, we again run the read-only workloads, initializing each index with 50M, 100M, 150M and 200M keys, respectively. As the results on all datasets are similar, we only report the results on FB in Fig. 1 (a). As the number of indexed keys increases, DILI maintains higher throughput than the alternatives. This indicates that DILI is more adaptive to the size of data.

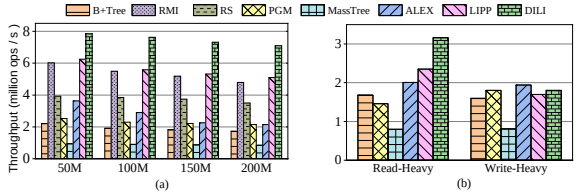

Figure 1: (a) Throughput on FB dataset with varying data cardinalities; (b) Performance with distribution shifting.

A.2 Effect of Distribution Shifting

To investigate if DILI still works well when keys from a different distribution are inserted, we design an experiment as follows. Suppose P_A is another pair set whose keys are in a different distribution from P. First, we build DILI and other indexes supporting inserts using their bulk loading algorithms on the whole P. Then, we repeatedly insert random keys from P_A via all indexes. Meanwhile, search operations are alternatively conducted. We tested a representative combinations of P and P_A from different distributions: FB and Logn. We observe each method's throughput on Read-Heavy and Write-Heavy workloads. Fig. 1 (b) reports that DILI performs a bit worse than ALEX on Write-Heavy workload. The reason behind is that DILI is built to grasp the distribution characteristics of specific dataset. Thus, inserting pairs from different distribution will incur more conflicts and result in adjustments happen more frequently. However, on Read-Heavy workload, DILI still has clear advantages over other alternatives. Also, in real-life scenarios, queries are often more common than insertions. Thus, DILI is supposed to have better performance than other state-of-the-art methods in practice.

A.3 Effect of Skewed Writes

R2.D3
M2

To investigate the performance of DILI and other alternatives with skewed writes, we design an experiment as follows. Suppose P, whose size is 100M, is the pair set used in the bulk loading stage. The range of $\text{KEYS}(P)$ is $[A, A+10\delta)$ and Q is another pair set whose keys are in a different distribution from P. We first compress the keys in $\text{KEYS}(Q)$ by mapping them to the range $[A, A+\delta)$. Next, a new pair set Q' is generated by randomly selecting 100M distinct mapped keys and packing them with the original record pointers of Q together. We use a Q with 200M elements to generate Q'. DILI, B+Tree, ALEX and LIPP are built using their bulk loading algorithms on the P. Afterwards, random pairs from the skewed set Q' are repeatedly inserted and then search operations are conducted via all indexes. We test all possible combinations of P and Q' from three different distributions: FB, WikiTS and Logn. We observe each method's throughput on read and write operations. Fig. 2 reports the results of DILI and the competitors.

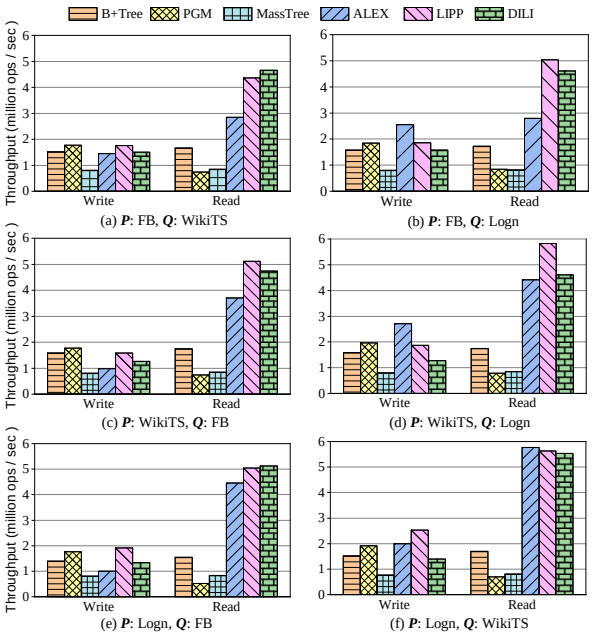

Figure 2: The effects of skewed insertions

Referring to Fig. 2, DILI does not have clear advantages over other competitors as before. It is because that DILI is a distribution-driven learned index which has customized node structure for each dataset. When the distribution of the inserted keys is skewed, more conflicts and node adjustments tend to happen, which result in a higher tree. For example, after inserting 100M skewed keys from the WikiTS dataset to the DILI built with the FB dataset, the average height of the DILI is 4.22. In contrast, if the inserted keys are not out of the initial distribution, the height is 3.86. Nevertheless, in comparison with LIPP and ALEX, DILI still achieves comparable or even better results.

A.4 Memory Cost Analysis in Write-heavy Workloads

We also conduct experiments to investigate the effect of insertions on the memory costs of DILI and the alternatives. Similarly, we build all indexes with half of the dataset and use them to carry out

R2.D5
M2
M5
M7

100M insertions where the inserted pairs are from the remaining half of the dataset. Table 1 shows the memory cost comparisons among all indexes.

Table 1: The memory costs (10^9 bytes) of different indexes in write-heavy workloads

Dataset	Before/after insertions	B+Tree Ω=32	MassTree	PGM	ALEX Γ=16MB	LIPP	DILI
FB	Before	1.56	1.56	1.60	2.35	12.68	5.07
	After	3.12	3.12	3.21	4.59	22.60	8.91
WikiTS	Before	1.56	1.56	1.60	2.31	10.11	3.95
	After	3.12	3.12	3.20	4.31	21.22	5.79
Logn	Before	1.56	1.56	1.60	2.30	14.53	3.38
	After	3.12	3.12	3.20	4.10	18.04	3.67

As shown in Table 1, the memory costs of B+Tree, MassTree and PGM are smaller than that of ALEX, DILI and LIPP. DILI achieves comparable results with ALEX. After the same 100M insertions from the Logn dataset, DILI uses less memory than ALEX. Compared to DILI, the memory costs of LIPP on all of the three datasets are much larger. This indicates that DILI is able to avoid much more conflicts as well as the empty slots in the entry array.

A.5 DILI vs RS

R1.R4 M2

We also conduct experiments to investigate why DILI outperforms RS. Similarly, the search with RS are broken down into two steps. At the first step, RS predicts for the search key a position through a linear spline and a radix table. Then, at the second step, RS conducts a local linear search to find the true position. Table 2 shows the breakdown time cost of DILI and RS.

Table 2: DILI vs RS

Dataset	Model	Step-1 (ns)	Step-2 (ns)	Total (ns)
FB	RS	264	41	305
	DILI	**75**	**75**	**150**
WikiTS	RS	212	52	264
	DILI	**75**	64	**139**
OSM	RS	181	**37**	218
	DILI	**74**	52	**126**
Books	RS	179	**31**	210
	DILI	**76**	77	**153**
Logn	RS	101	31	132
	DILI	**73**	**43**	**116**

Referring to Table 2, the average response times of the first step with RS are larger than the counterpart with DILI. At the second step, RS additionally needs 41 ns, 52 ns, 37 ns, 31 ns and 31 ns on average to carry out the local search on the five datasets, respectively, whereas the response times of DILI at the second step are smaller or comparable. Thus, RS's query performance is overall worse than DILI.

A.6 Frequency and Effect of Node Adjustments

R2.D4 M2

To investigate the effects of the adjusting strategy, we conduct experiments using DILI and its variant DILI-AD which does not adopt the adjusting strategies when performing insertions. Following the Write-only workload evaluation process in Section **??**, both indexes are first built with half of the datasets. Then, for each dataset, we use both indexes to sequentially execute a Write-only workload containing 100M insertions and a Read-only workload containing 100M queries. The comparisons of their response times per insertion, memory costs, average heights, lookup time after insertions are shown in Table 3 below. In particular, the third column of this table shows the average number of insertions per adjustment with DILI, which is the frequency of adjustments in DILI's leaf nodes.

Table 3: The effects of the adjusting strategy

Dataset	Model	Avg # of insertions per adjustment	Insertion time (ns)	Memory cost (10^9 bytes)	Avg height	Lookup time (ns)
FB	DILI-AD	-	**442**	9.31	3.91	187
	DILI	229.1	486	**8.91**	**3.86**	**180**
WikiTS	DILI-AD	-	**349**	5.86	3.58	172
	DILI	758.4	361	**5.78**	**3.55**	**168**
Logn	DILI-AD	-	**304**	3.71	3.11	130
	DILI	1304.6	310	**3.67**	**3.09**	**126**

Apparently, the adjusting strategy in DILI's insertions will result in longer insertion time as it sometimes requires extra operations to collect all pairs covered by a node and train a new linear regression model. However, the adjusting strategy makes DILI avoid more conflicts such that DILI will have a shorter structure and achieves better lookup time and memory costs.

A.7 A Possible Way to Reduce DILI's Construction Time

R4.R3 M2

The most time-consuming step in DILI's construction is the greedy merging algorithm for getting the BU nodes at the bottom layer. Because we need to solve a MSE minimization problem for the union of each continuous two pieces, *i.e.* $\mathcal{I}_u^k \bigcup \mathcal{I}_{u+1}^k$ and calculate the MSE to select the next two pieces to merge. Also, after merging $\mathcal{I}_u^k$ and $\mathcal{I}_{u+1}^k$ into a new piece, namely $\tilde{\mathcal{I}}_u^k$, we also need to conduct similar operations for $\mathcal{I}_{u-1}^k \bigcup \tilde{\mathcal{I}}_u^k$ (Algorithm **??**).

A direct yet effective approach to make the BU node generation more efficient is sampling. When a piece covers many keys, we could randomly or selectively sample part of the keys, *e.g.*, select one key out of two, to get the linear regression model and calculate the cost. The sampling strategy makes little influence on the whole BU-tree node layout and the performance of the generated DILI. However, it will clearly reduce the construction time of the BU-tree and DILI. An experiment is conducted to show this.

We apply the sampling strategy on the BU-tree's construction as follows. When a piece $\mathcal{I}_u^k$ covers more than 8 keys, we only use half keys get the linear regression models. A DILI is then built based on this BU-tree. As a comparison, we build another DILI without adopting the sampling strategy. According to our statistics, the construction time of the former DILI is 1 minute more less than that of the latter. Their comparison results on the lookup time are shown in Table 4. Apparently, the lookup time of the DILI with sampling is only slightly larger than that of the ordinary DILI.

Table 4: The lookup time of DILI variants

Method	FB	WikiTS	OSM	Books	Logn
DILI-W-Sampling	160	145	121	154	117
DILI	150	139	117	148	116

Considering the construction process of a DILI will be executed once only, a slightly higher construction time is acceptable.

A.8 Concurrent Insertions and Deletions in DILI

R4.R6 It is possible that DILI support concurrent data updates. It is noteworthy that either an insertion or deletion operation involves only one leaf node. The node adjustment of DILI is much simpler than the rebalance operation of the B+Tree. Theoretically, the lock-free and lock-crabbing approaches can also be applied to DILI, in the same way as how they are applied to the B+Tree. For example, the lock-crabbing protocol for an insertion can be simply applied to DILI as follows:

(I) Get the lock for the lowest leaf node, namely N_D, covering the pair to be inserted.

(II) If N_D has an empty slot for the pair, put the pair here and release the lock for N_D.

(III) If a conflict happens, the following steps are sequentially executed:

1) If a node adjustment is required, carry out the adjustment and exit. Otherwise, execute the following operations.
2) Create a new leaf node N'_D and put the node at the conflicted slot.
3) Get the lock for N'_D.
4) Release the lock for N_D.
5) Store the conflicting pairs in N'_D.
6) Release the lock for N'_D.

The lock-crabbing protocal for a deletion is similar:

(I) Get the locks for the lowest leaf node and its parent node, namely N_D and N_P, respectively, covering the key to be deleted.

(II) Delete the corresponding pair from N_D.

(III) If N_T is an internal node or N_D covers more than one pairs, release the locks for N_D and N_P, and exit. Otherwise, execute the on the following operations:

1) Suppose $N_P.\boldsymbol{V}[k]$ stores the pointer to N_D. Set $N_P.\boldsymbol{V}[k] = p$.
2) Delete the node N_D.
3) Release the lock for N_P and exit.