SALI: A Scalable Adaptive Learned Index Framework based on Probability Models

A COMPREHENSIVE EVALUATION RESULTS

A.1 Throughput

Figure 1-4 showcase the complete evaluation results for all datasets and different read/write ratios (uniform distribution), regarding the complete assessment of throughput as mentioned in the main text. SALI exhibits superior throughput compared to other SOTA learned indexes.

A.2 Tail latency

Figure 5-8 present the comprehensive evaluation results for all datasets and various read-write ratios (uniform distribution), regarding the complete assessment of tail latency as mentioned in the main text. SALI maintains low tail latency across the board.

B ANALYSIS OF THE LIMITATIONS OF SALI

B.1 The limitation of read-evolving strategy

The insight 4 in Section 4.3.1 states that flattened tree structures under easy datasets do not evolve. In such cases, enabling the read-evolving strategy would introduce additional overhead to determine if a target node is a hot node, which would compromise the read performance of the SALI.

Specifically, we analyze the overhead through experimental evaluation. When SALI activates read-evolving, the overhead during lookups on a read-only workload (uniform distribution) with different datasets is illustrated in Figure 9. There are three types of overhead during lookups: 1) the if statement to determine whether the lookup triggers probability calculation, where each thread calculates the probability every 10 lookups, 2) probability calculation, and 3) determining whether a node is a hot node. Note that, for better clarity in illustrating the performance bottlenecks, all nodes are non-evolving (manually controlled) as node evolving may not always be applicable due to the small tree depth of certain datasets.

In Figure 9, $SALI + Prob.$ represents the throughput of SALI with the first and second types of overhead, while $SALI + R.evol.$ represents the throughput of SALI with all three types of overhead. *SALI* represents the read-evolving switch is turned off. It can be observed that the first two types of overhead are negligible, and the main overhead comes from the third type.

Therefore, when enabling the read-evolving strategy in easy datasets, the internal nodes of SALI will not undergo evolution, and there will be an additional overhead in determining whether a node is a hot node, potentially leading to a loss of lookup performance. We acknowledge that evaluating the benefits of enabling read-evolving in specific data distributions can be challenging, and this will be a focal point of our future work.

Figure 10 illustrates the throughput of the index when read-evolving is enabled under a read-only workload (uniform distribution). In contrast to Figure 9, the $SALI + R.evol.$ configuration includes node evolving in the OSM and GENOME datasets, while the experimental results for the COVID and LIBIO datasets align with those in Figure 9. This is because the flattened tree structures in easy datasets do not undergo evolution (refer to Insight 4 in Section 4.3.1).

B.2 The limitation of supporting duplicate data

Currently, SALI does not support duplicate keys. The reason for this is that SALI, being based on the LIPP structure, does not currently support the insertion of duplicate data. However, Wu et al. [1] have suggested that it is relatively straightforward for indexes to accommodate duplicate keys, such as by maintaining a pointer to an overflow list. As part of our future work, we plan to focus on implementing the insertion of duplicate data.

C BUFFER-BASED AND MODEL-BASED INSERTION STRATEGY

In this section, we will provide a comprehensive discussion on the strengths and weaknesses of the buffer-based insert strategy (out-of-place insert) and the model-based insert strategy (in-place insert). Our aim is to emphasize the significance of choosing different index structures based on specific scenarios.

C.1 buffer-based insertion strategy

The major performance limitation of the buffer-based insertion strategy lies in its significant lookup error. Unlike the model-based strategy, it lacks the capability to adjust the distribution of stored keys through reserved gaps, which could result in a linearized index structure and reduce or eliminate errors (i.e., ALEX, LIPP). The larger lookup error adversely affects the lookup and insertion performance of the index.

From a more detailed perspective, there are two ways to insert data in the buffer. The first approach involves maintaining the insertion order. While this method is more efficient for searching, it requires moving many keys to maintain the order during insertion. As a result, it degrades the insertion performance of the buffer. The second approach is the append-only insertion method, where there is no need to maintain the insertion order. This method is efficient for write operations but introduces additional overhead during searching and retraining processes. Regardless of the approach chosen, both searching and inserting operations in the buffer result in significant time overhead, leading to suboptimal index performance.

However, the buffer-based insertion strategy is not without its opportunities. It demonstrates significant advantages in the following two scenarios: 1) PGM, a type of learned index similar to LSM-Tree, exhibits remarkable performance benefits in write-only workloads. In such scenarios, the buffer-based insertion strategy proves highly effective. 2) In certain environments with strict space

Conference SIGMOD '24, June 11–06, 2024, Santiago, Chile
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

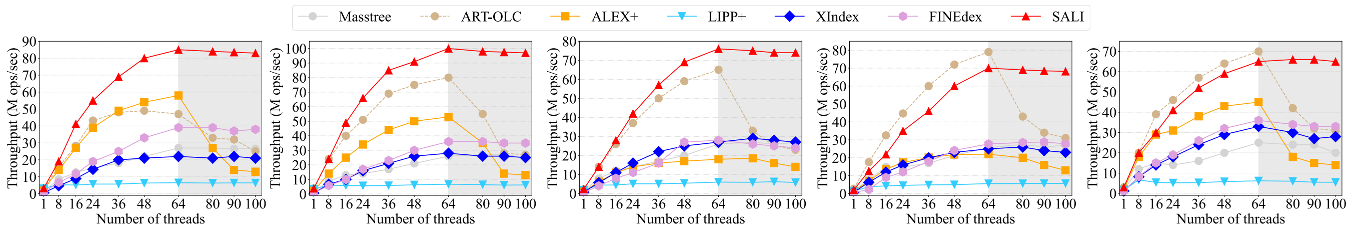

Figure 1: The indexes scalability on write-only workloads (Throughput).

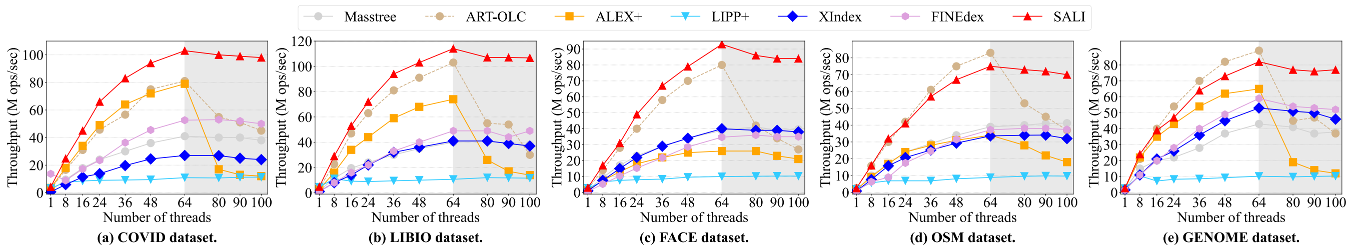

Figure 2: The indexes scalability on balance workloads (Throughput).

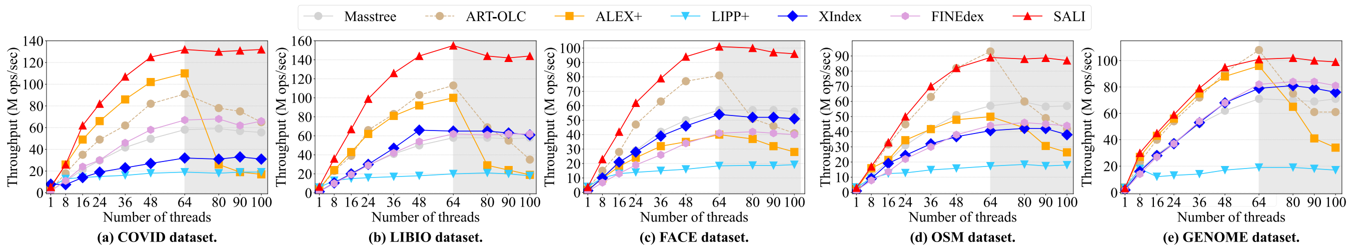

Figure 3: The indexes scalability on read-intensive workloads (Throughput).

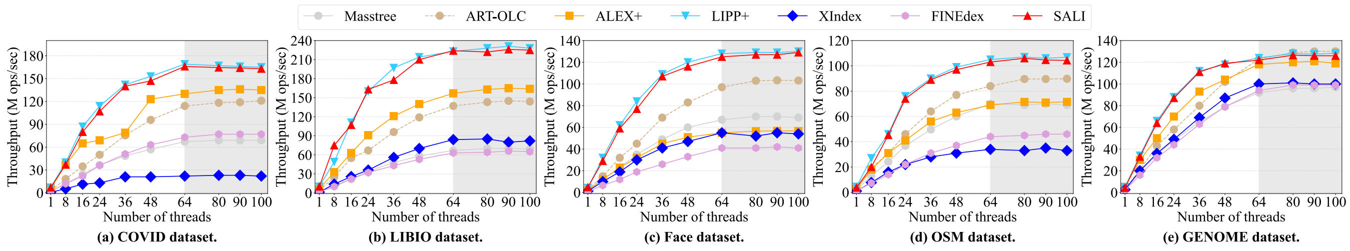

Figure 4: The indexes scalability on read-only workloads (Throughput).

limitations, such as embedded systems, the buffer-based insertion strategy's corresponding compact storage (i.e., without gaps) offers space savings of two orders of magnitude compared to B+ trees. Such learned indexes (e.g., PGM) are more suitable for these application scenarios.

In summary, while the buffer-based insertion strategy may have limitations, it demonstrates substantial advantages in specific scenarios such as write-only workloads and environments with stringent space constraints.

C.2 model-based insertion strategy

The model-based insertion strategy offers superior performance in learned indexes due to the following reasons:

1)This strategy introduces gaps between stored keys, altering the cumulative distribution of the stored keys to approximate a linear distribution. As a result, the index structure can be approximated as a linear model with minimal error (i.e., ALEX). Additionally, techniques like chaining in designs such as LIPP enable conflict resolution, ensuring precise lookups and significantly improving query performance. 2)Unlike the buffer-based insertion strategy, the model-based approach reserves gaps that allow for efficient insertion of new keys without the need to 'shift' many existing keys. This substantially enhances the insertion performance of the index.

In summary, the model-based insertion strategy in learned indexes delivers improved performance through CDF optimization, precise lookups, and efficient insertions facilitated by reserved gaps. Therefore, in the application scenarios of this paper, we have opted to evaluate SALI using the model-based insertion strategy.

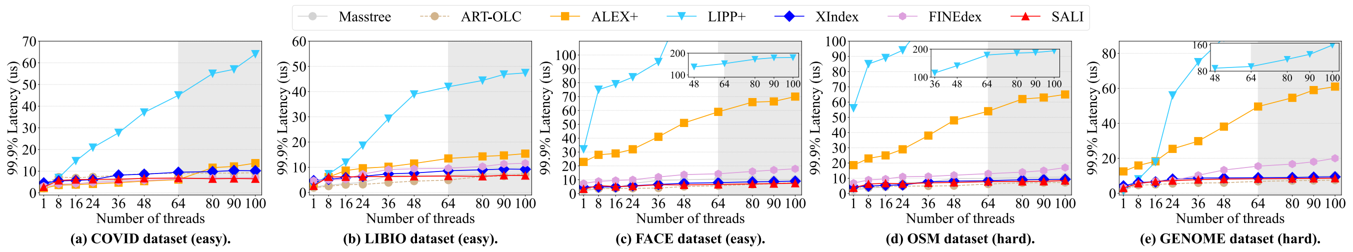
Figure 5: The indexes scalability on write-only workloads (Latency).

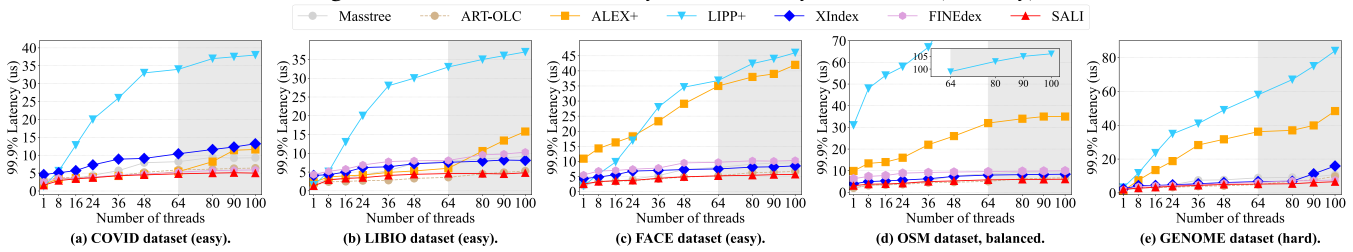
Figure 6: The indexes scalability on balance workloads (Latency).

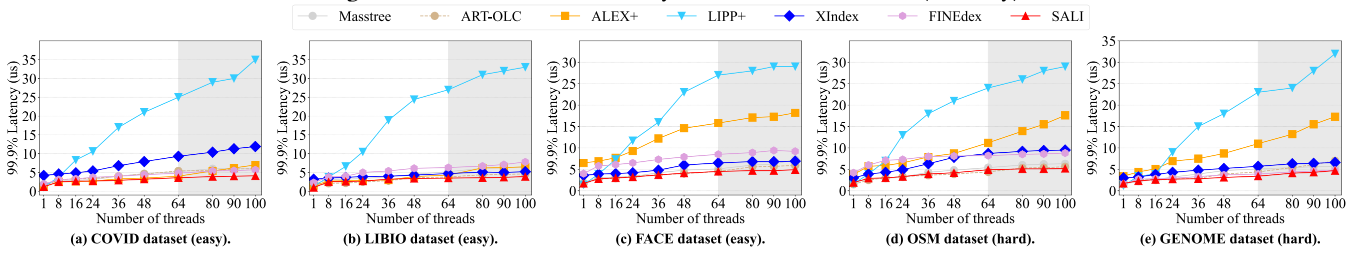
Figure 7: The indexes scalability on read-intensive workloads (Latency).

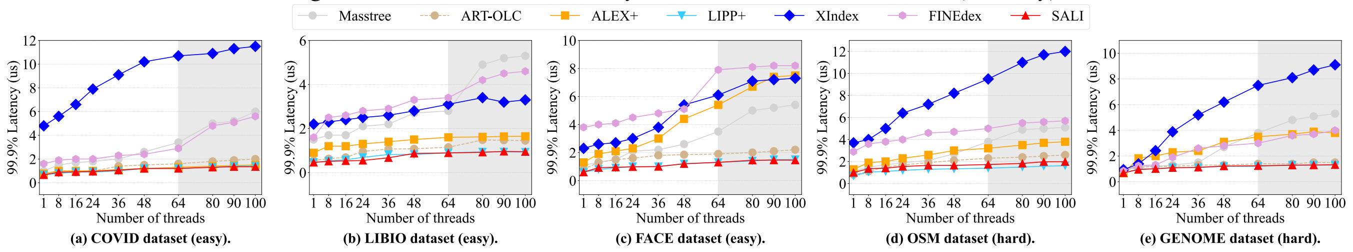
Figure 8: The indexes scalability on read-only workloads (Latency).

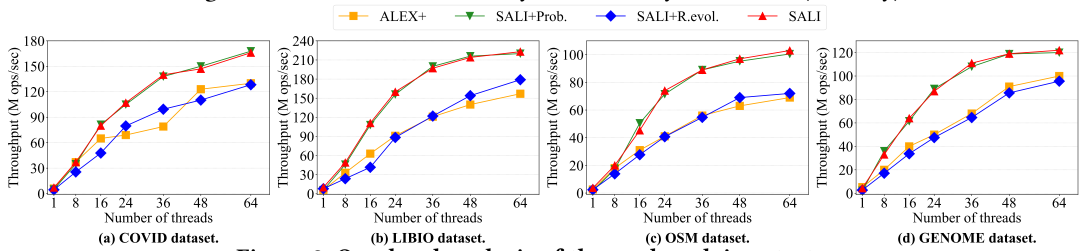
Figure 9: Overhead analysis of the read-evolving strategy.

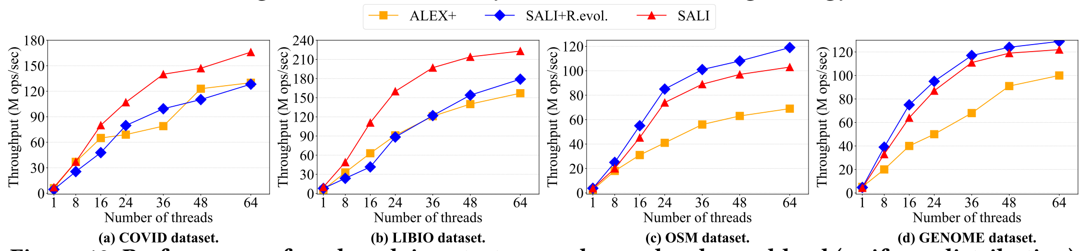
Figure 10: Performance of read-evolving strategy under read-only workload (uniform distribution).

REFERENCES

[1] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.