A APPENDIX

A.1 Causes of ALEX's Space Overhead Surge and Crash

The underlying cause of ALEX's drastic space expansion (Figure 16(a)) or even immediate crashes (Figure 16(b)) can be explained as follows. Since insert operations in learned indexes are required to maintain strict key ordering, insert conflicts or increased query errors after insertion significantly raise the cost of inserts. To mitigate this cost, the index triggers internal node splits. When there is a significant difference in the distribution of inserted data, two serious situations can occur:

(1) **In Scenario 1,** to accommodate keys with significantly varying numerical ranges within a unified structure, ALEX expands its root node until it fully covers the entire key domain. Specifically, ALEX's internal nodes employ a strategy of evenly partitioning the key value domain to ensure precise lookups. For example, if the value domain is [0-20], it can be divided into four slots: [0–5], [5–10], [10–15], and [15–20]. If keys with substantially different distributions are now inserted, e.g., inserting key=1000, the root node will expand the domain size using a power-of-2 expansion factor to encompass key=1000. Thus, it is necessary to compute $20 \times 2^n > 1000$, yielding the minimum n as 6, corresponding to the expanded domain of [0-1280]. After expansion, the root node's slots will increase from the original 4, extending with the same stride to cover the new domain [0-1280], resulting in a total of 256 slots ($1280/5 = 256$), which severely enlarges the index's storage footprint. Consequently, when ALEX inserts a small number of keys with large numerical differences, this process leads to substantial space overhead (see Figure 16(a)).

(2) In Scenario 2, the insertion of duplicate keys persistently triggers shifting operations, leading to the accumulation of errors in leaf nodes. This increase in shift frequency and error buildup causes the insertion cost in ALEX's leaf nodes to progressively escalate, ultimately initiating SMO operations, i.e., node splitting. However, the nodes employ a splitting strategy that evenly partitions the key domain, which fails to effectively segregate duplicate keys. Consequently, the insertion cost cannot be reduced through node splitting, resulting in repeated triggering of splitting operations. This, in turn, induces cascading splits from leaf nodes to internal nodes, with the number of nodes growing exponentially and ultimately leading to memory overflow (see Figure 16(b)).

Specifically, Figure 19 illustrates the reason why ALEX crashes when inserting a small number of duplicate keys. Since insertions in learned indexes must maintain ordering, insertion conflicts incur numerous shift operations, thereby causing substantial insertion costs. To mitigate insertion costs, ALEX performs node splitting operations. ALEX's splitting adopts a simple approach of evenly dividing the key range within the node; that is, if the keys in a node range from [0, 100], the split results in two nodes covering [0, 50] and [51, 100].

As shown in Figure 19, if $key = 42$ is repeatedly inserted into ALEX hundreds of times, this splitting method struggles to quickly isolate the duplicate keys into a single node with only a few splits, resulting in continuous splitting and failing to effectively reduce the insertion cost. ALEX performs recursive (cascading) lateral splits; the size of the internal node pointer array grows exponentially until it reaches the maximum internal node size or ultimately causes an OOM error.

Consequently, ALEX fails to effectively handle keys with significant distribution variations, which stems from its rigid key-range partitioning mechanism that equally splits the key space. This structural flaw severely constrains its time-space tradeoff efficiency while substantially compromising robustness and practical applicability.

A.2 Causes of DILI's Crash

Our research reveals that crashes in the DILI structure have two main causes:

(1) The structure employs hard-coded definitions for key types, lacking adaptability to different key types, which renders it incapable of handling key values that exceed the preset range. For instance, it can process the Libio dataset with a smaller key value $(2^{14}, 2^{29})$, but fails to adapt to the Osm dataset with a larger key value $(2^{54}, 2^{64})$. Due to the constraints of this hard-coded implementation, extending the supported key type range would require modifying all relevant type definitions in the source code, which we did not undertake.

(2) The linear fitting model employed by this structure exhibits insufficient resolution for densely distributed key values, resulting in multiple adjacent keys being mapped to the same position. In such cases, the absence of an effective conflict resolution mechanism (at the bottom-level storage nodes) further leads to query errors.

Therefore, when designing learned indexes, template classes should be used to enable adaptive key type support, and discriminative precision should be fully considered in linear fitting models to avoid query errors in dense key distribution regions.

A.3 Causes of FINEdex's Crash

The primary cause of structural collapses in FINEdex lies in the design flaws of its structural modification operation (SMO) strategy.

Specifically, as illustrated in Figure 20, when the level bin associated with a leaf node becomes full, it transforms into a learned node that employs a linear model for key retrieval (Figure 20(b)). If the number of learned nodes connected below the leaf node exceeds a threshold, a leaf-node-level SMO is performed, which involves resorting all keys contained in that leaf node and splitting them into new learned nodes. However, under sequential insertion workloads, newly inserted keys are only written to the rightmost level bin of the leaf node. With continuous sequential insertions, only the last level bin fills up and converts to a linear model node (Figure 20(a) and (b)). This results in the continuous allocation of new buffers below the leaf node's associated linear model node to accommodate subsequent insertions, leading to persistent tree height growth (Figure 20(c) and (d)). As the preset threshold for triggering SMO is never exceeded, the leaf-node-level SMO remains untriggered, causing the tree to grow increasingly taller and ultimately leading to structural collapse.

In summary, the structural collapse of FINEdex is rooted in its SMO logic's inability to adapt to diverse workloads. Consequently, index design must thoroughly account for its performance under varying workload conditions.

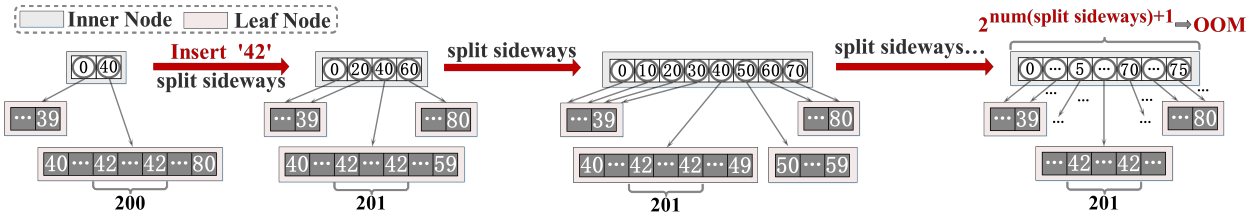

Figure 19: Space algorithmic complexity attack on inner nodes of ALEX.

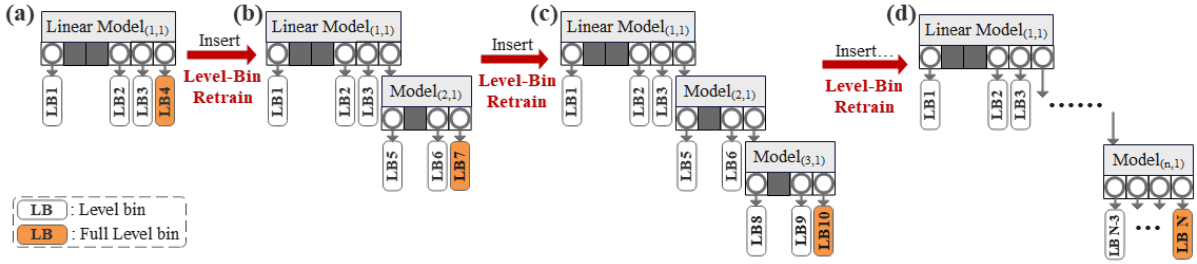

Figure 20: Schematic Illustration of the Causes of Crashes in FINEdex during Sequential Insertion.

Table 6: Assumed Prices of Different Time-Space Configurations under Various γ Values.

	1 GB					2 GB					3 GB					m GB				
	$\gamma=0.39$	$\gamma=0.52$	$\gamma=0.75$	$\gamma=0.93$	$\gamma=1$	$\gamma=0.39$	$\gamma=0.52$	$\gamma=0.75$	$\gamma=0.93$	$\gamma=1$	$\gamma=0.39$	$\gamma=0.52$	$\gamma=0.75$	$\gamma=0.93$	$\gamma=1$	$\gamma=0.39$	$\gamma=0.52$	$\gamma=0.75$	$\gamma=0.93$	$\gamma=1$
1 hour	k	k	k	k	k	$1.31k$	$1.42k$	$1.68k$	$1.95k$	$2k$	$1.47k$	$1.75k$	$2.37k$	$2.70k$	$3k$	$m^{0.39}$k	$m^{0.52}$k	$m^{0.75}$k	$m^{0.93}$k	m^{1}k
2 hours	$2k$	$2k$	$2k$	$2k$	$2k$	$2.63k$	$2.85k$	$3.36k$	$3.90k$	$4k$	$2.94k$	$3.50k$	$4.73k$	$5.40k$	$6k$	$2\cdot m^{0.39}$k	$2\cdot m^{0.52}$k	$2\cdot m^{0.75}$k	$2\cdot m^{0.93}$k	$2\cdot m^{1}$k
3 hours	$3k$	$3k$	$3k$	$3k$	$3k$	$3.94k$	$4.27k$	$5.04k$	$5.84k$	$6k$	$4.41k$	$5.25k$	$7.10k$	$8.09k$	$9k$	$3\cdot m^{0.39}$k	$3\cdot m^{0.52}$k	$3\cdot m^{0.75}$k	$3\cdot m^{0.93}$k	$3\cdot m^{1}$k
n hours	nk	nk	nk	nk	nk	$1.31nk$	$1.42nk$	$1.68nk$	$1.95nk$	$2nk$	$1.47nk$	$1.75nk$	$2.37nk$	$2.70nk$	$3nk$	$n\cdot m^{0.39}$k	$n\cdot m^{0.52}$k	$n\cdot m^{0.75}$k	$n\cdot m^{0.93}$k	$n\cdot m^{1}$k

The gray-shaded entries in the table represent the market pricing models adopted by major cloud service providers.

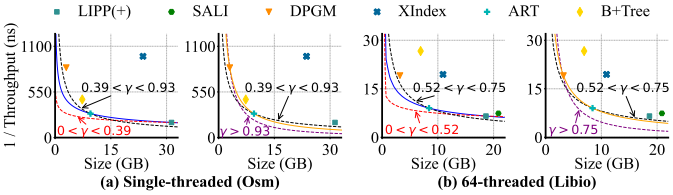

Figure 21: Index recommendations under varying γ.

A.4 Time-Space Pricing Under Different γ Configurations

Figure 21 illustrates two representative scenarios where LIPP becomes the recommended choice. In the single-threaded environment under the Osm dataset, LIPP achieves the lowest time-space cost when $\gamma < 0.39$, making it the most economical option under this pricing scheme.

Table 6 presents the time-space leasing prices under different hypothetical γ configurations. When $\gamma = 1$, the pricing scheme charges k RMB per hour per GB. For example, leasing 1 GB 1 hour costs k RMB, while n hours with m GB costs $n \times m \times k$ RMB. When $\gamma = 0.39$, the pricing follows k RMB per hour per GB^{γ}. Under this model, 1 GB 1 hour remains k RMB, but 1 hour with 2 GB costs $1.31k$ RMB, and m hours with n GB costs $m \times n^{0.39} \times k$ RMB. Here, the effective memory price becomes significantly lower than market rates. LIPP emerges as the winner in single-threaded environments on the Osm dataset only when memory is priced even lower than the rate corresponding to $\gamma = 0.39$. Actual prices corresponding to other γ values in Figure 21 are provided in Table 6.