ALISA: An Adaptive Learned Index Structure for Spatial Data on Solid-State Drives

Che-Wei Lin
Department of Computer Science, National Yang Ming Chiao Tung University
lincw.cs11@nycu.edu.tw

Chun-Feng Wu
Department of Computer Science, National Yang Ming Chiao Tung University
cfwu417@cs.nycu.edu.tw

ABSTRACT

Spatial learned index is becoming popular as a solution to relieve the intense storage demands and high I/O costs of spatial databases. LISA, the original and most prominent spatial learned index structure, is tailored for HDD-resident spatial data and comes with strict data arrangement requirements. Given that direct SSD migration may drastically impair SSD durability especially when page utilization is low, this work aims to adapt this innovative index structure to SSDs to leverage the faster performance and expand application possibilities. We propose an Adaptive Learned Index structure for Spatial dAta on SSDs (ALISA), with mechanisms to persistently monitor updated data distribution and adaptively restructure to align with SSD access characteristics. The evaluation results show that ALISA can significantly extend SSD lifespan and improve low page utilization, thereby enhancing query performance.

ACM Reference Format:
Che-Wei Lin and Chun-Feng Wu. 2024. ALISA: An Adaptive Learned Index Structure for Spatial Data on Solid-State Drives. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676677

1 INTRODUCTION

Spatial indexes are widely adopted in spatial databases to efficiently manage multi-dimensional data and improve query processing [14]. As spatial data grows in size and variety, traditional spatial indexes face challenges with high space and time demands for indexing such large datasets [26]. LISA [18] is the original and most notable spatial learned index structure employing learned index technology to address these challenges. Since LISA is designed for HDD-resident spatial data, its direct migration to SSDs can greatly reduce storage device durability, and it becomes even worse with low page utilization. To extend SSD lifetime by reducing the number of flash page writes, this work aims to extend LISA to adaptively manage spatial key-value data with considering the SSD's constraint by advocating a middleware between LISA and SSDs.

In recent years, a vast amount of location-based data has emerged from an array of GPS-enabled devices, including mobile devices, IoT sensors, and wearable devices [5, 12]. The rapid generation of location-based data has spurred significant research efforts aimed at developing optimized spatial databases [20], which often leverage advanced hardware and integration techniques to meet the growing demands of contemporary applications [8, 15, 25]. Meanwhile, NAND flash-based Solid-State Drives (SSDs) are extensively utilized as secondary storage in aforementioned devices. Their enhanced features, including lightweight, small size, shock resistance, portability, and faster access speeds, set them apart from Hard Disk Drives (HDDs) [16, 21, 31], especially in mobile devices and embedded systems. Many studies focusing on integrating spatial data with SSDs [4, 13, 24, 29] have highlighted the desire to shift the substantial space demands of spatial databases from HDDs to SSDs.

To enhance spatial query efficiency, many studies have designed elaborate spatial indexes to reduce data retrieval volume [1, 2, 11]. Nonetheless, these traditional spatial indexes have some limitations, such as the inability to be fine-tuned for specific datasets [23], and they can become both space and time-consuming as data volumes grow due to the increased number of nodes for storing metadata. For example, R-tree [11] is the most renowned spatial index and widely used in commercial database such as PostgreSQL [22] and MySQL [9]. A larger amount of data results in a larger R-tree, leading to more I/O costs during queries and increased space required to store the Minimum Bounding Box, the additional node information of the R-tree. Recent studies [7, 10, 17] have introduced learned indexes, which employ machine learning models to efficiently predict data indexes and reduce storage consumption by only storing model parameters. Among them, LISA is the original and the most notable learned index structure tailored for HDD-resident spatial data. LISA transforms multi-dimensional spatial data into one-dimensional ordered values and computes the necessary index to minimize I/O cost. Compared with other multi-dimensional learned index structures [23, 28], LISA stands out by supporting update operations and being specifically designed for disk-based databases. Relative to traditional spatial index structures such as R-tree, LISA consumes less storage space and reduces I/O operations when processing range and KNN queries [18].

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM 979-8-4007-1077-3/24/10…
https://doi.org/10.1145/3676536.3676677

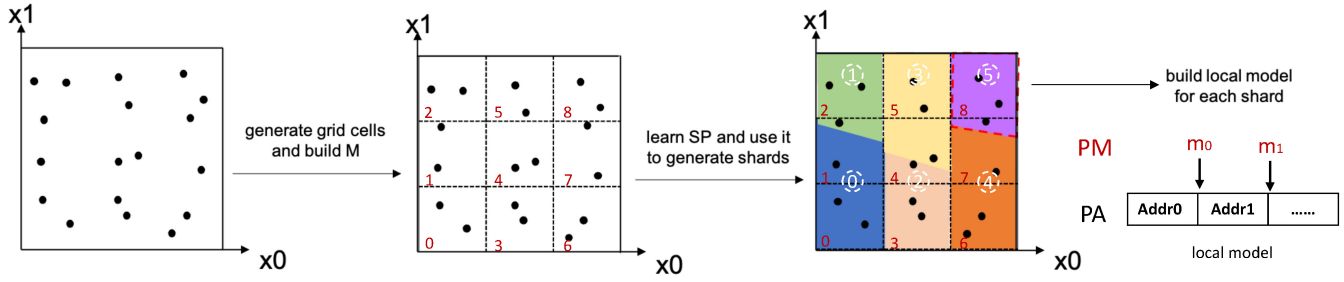


Figure 1: LISA Construction Process

Meanwhile, SSDs have become a favored alternative to HDDs across numerous applications, benefiting from their performance improvements and cost reductions year-over-year [19, 27, 30, 34]. We aim to integrate LISA with SSDs to provide better spatial data management performance. Based on our observation, the access behavior gap between LISA and SSDs may increase the number of written flash pages during data updates, thus significantly impacting SSDs' lifetime. This gap arises because LISA must update data in compliance with SSDs' out-of-place constraint while also adhering to the neighboring-keys clustering constraint, which is a data arrangement requirement for LISA. Consequently, LISA is restricted to adopting Read-Modify-Write (RMW) operations for updating data on SSDs. Since the minimum RMW granularity must align with an SSD page size, the SSD reads out and writes back a whole page even when the updated data is smaller than the page size. This update-amplification issue severely affects SSD's lifetime, as the size of spatial data (usually several bytes) is much smaller than an SSD page (usually 4KB).

To mitigate the lifetime issue, this paper proposes a middleware to extend LISA to adaptively manage spatial key-value data while considering the SSD's write constraint. In particular, the proposed middleware comprises three components: (1) the adaptive update-interval model expander to minimize the number of written pages by keeping track of update interval, (2) the adaptive model compactor to minimize the number of written pages while also making pages as compact as possible, and (3) the model-aware data pinner to reduce unnecessary writes by pinning data of the least updated pages in the write buffer. The evaluation demonstrates that ALISA effectively enhances the lifetime of SSDs by reducing approximately 32% and 37% of pages written to the SSD compared to LISA and R-tree, respectively. Moreover, ALISA achieves faster execution times, completing range queries approximately 15% and 19% faster than LISA and R-tree, respectively, attributed to the improved page utilization.

The rest of this paper is organized as follows: Section 2 presents the background, observation, and motivation. In Section 3, we extend LISA to ALISA to enhance SSD's lifetime and the performance of range queries. Section 4 provides the analysis and experimental results. Section 5 concludes this work.

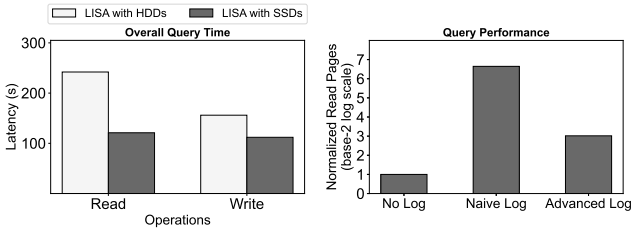
2 BACKGROUND, OBSERVATION, AND MOTIVATION

2.1 Background

2.1.1 Learned Index Structure for Spatial Data (LISA). LISA stands as a spatial learned index solution aimed at efficiently indexing multi-dimensional key-value data. Unlike traditional tree-based or hash-based data structures that typically require substantial storage space to accommodate corresponding data structures, LISA computes the necessary index with low memory consumption. To achieve this, LISA transforms multi-dimensional spatial data into one-dimensional ordered values and builds a computational model mapping these ordered values to physical addresses.

Technically, LISA comprises four parts: the grid cells, the mapping function M, the shard prediction function SP, and the local model for each shard. The construction process is illustrated in Figure 1. Initially, LISA partitions the space into grid cells following the data distribution along the axes, ensuring each cell contains similar numbers of key-value data, and numbers these cells for further computation. LISA then constructs a partially monotonic mapping function, denoted as M, which converts multi-dimensional keys into one-dimensional mapped values. This conversion is based on coordinates and cell IDs. Specifically, keys associated with larger cell IDs are mapped to correspondingly larger values compared to those linked with smaller cell IDs. Furthermore, within a cell, keys having a larger Lebesgue measure than the lower-left boundary are assigned greater mapped values.

Once the keys' mapped values are calculated, LISA learns a monotonic shard prediction function SP comprises a series of piecewise linear functions to assign shard id to every mapped value. The training phase ensures that keys are evenly spread across shards, and keys within the same shard are stored on one or more consecutive pages, whereas keys from different shards must be stored on different pages. LISA constructs



(a) Latency Comparison. (b) LISA's Read Performance

Figure 2: SSDs Outperforms HDDs & LISA's Read Penalty after Log Updates

a local model for each shard to easily position key-value data within a shard. The local model consists of *PA* and *PM*. *PA* is a list comprising addresses for all pages encompassed within a shard. *PM* is a sequence of ascending mapped values allocated to represent each page in a shard. If a shard contains multiple pages, the placement of key-value data on each page should follow the "neighboring-keys clustering constraint". It means that keys must be strictly clustered together and located on the same page if their mapped values fall within the range of two consecutive *PM* values. Take Figure 1 for example, if $m_k = M(k)$ and $m_0 <= m_k < m_1$, then key k must be stored in page on $Addr_1$.

2.1.2 Solid-State Drives (SSDs). NAND flash-based SSDs have been widely used in computer systems due to their superior features over HDDs, including performance, portability, and energy efficiency. In the rest of the paper, we will refer to NAND flash-based SSDs simply as SSDs. There's a significant difference in how SSDs and HDDs update data. HDDs can perform in-place updates by erasing old data and writing new data to the same location. Such an approach in SSDs would cause considerable performance and energy waste, given that the smallest read/write unit is a page (usually 4KB) whereas the smallest erase unit is a block, each consisting of thousands of pages. To avoid this, data updates in an SSD shall follow the out-of-place update constraint via a Read-Modify-Write (RMW) procedure. That is to read out the targeted page, update it with new data, write it back to a new location, and then mark the original page as invalid.

However, updating only a few bytes to a flash page still requires rewriting the entire page, which severely hurts the lifetime because of SSDs' limited write endurance. While systems typically allocate some DRAM as a write buffer to accumulate updates, this does not significantly alleviate the lifetime issue, especially when the page accommodates tiny data, like key-value data. Systems usually adopt the log update [6, 19, 33] to alleviate the lifetime issues caused by updating key-value data on SSDs. Log update accumulates updated key-value data and writes them to some new pages

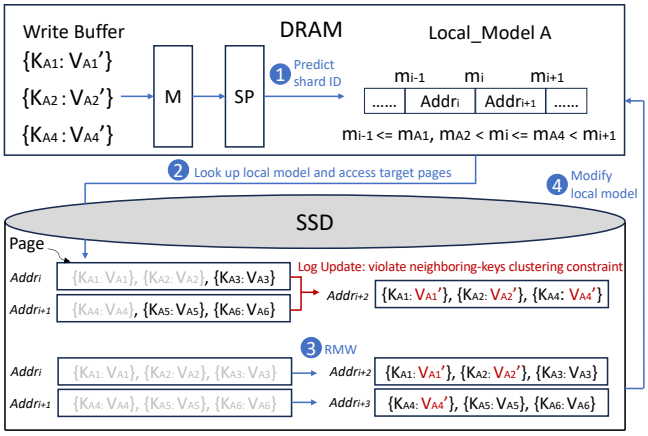


Figure 3: LISA Value Update Flow

without doing RMW. We illustrate a toy example in the top half inside SSD in Figure 3. Assuming to update three key-value data, that is $\{K_{A1} : V_{A1}\}$, $\{K_{A2} : V_{A2}\}$ in the page on address $Addr_i$ and $\{K_{A4} : V_{A4}\}$ in the page on $Addr_{i+1}$, all these data will be written to a new page on $Addr_{i+2}$ via log update.

2.2 Observation & Motivation

In key-value store, SSDs are configured as HDDs' cache or become a favored alternative to HDDs, due to its better performance on support random accesses and a lower price per byte year-over-year. As illustrated in Figure 2(a), running LISA on SSDs outperforms its performance on HDDs for both read and write operations. However, this discrepancy is not due to LISA being specifically optimized for SSDs; rather, it stems from the significantly faster access latency of SSDs compared to HDDs. In fact, LISA is primarily designed for HDD-resident spatial databases, making its design inherently incompatible with SSDs. To satisfy both the out-of-place update and neighboring-keys clustering constraints, LISA cannot adopt the log update method but is constrained to use the RMW operation.

We provide a toy example inside the SSD in Figure 3 to explain that the log update violates the neighboring-keys clustering constraint. The key-value data $\{K_{A1}: V'_{A1}\}$ denotes that the value of the first K-V item in shard A is updated to V'_{A1}. LISA identifies the shard IDs of the updated data with *M* and *SP* (step 1), then looks up the corresponding local models to obtain the page addresses (step 2). The upper part of the SSD illustrates the log-update method. Assuming $m_{A1}<m_{A2}<m_{A3}$ and $m_{A4}<m_{A5}<m_{A6}$, this approach violates LISA's neighboring-keys clustering constraint because mapped value interval of new pages (m_{A1} to m_{A4}) overlaps with the intervals on $Addr_i$ (m_{A3}), leading the address of the new log page cannot be inserted into the local model. As a

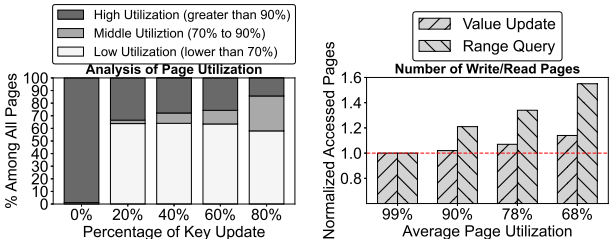
result, the log page can only be stored in the extra log region, resulting in a significant read penalty due to the necessity of traversing the entire log region in every read operation.

We demonstrate the long-term read penalty when LISA adopts log update in Figure 2(b). For the naive log region group, all updated data will be appended in the unified log region. For the advanced log region group, all updated data will be distributed in their corresponding log regions according to shard ID. After flushing the updated data into the SSD, our emulator executes ten thousand range queries and keeps track of the total pages read from the SSD. The result shows that LISA reads up to 50 times the number of pages with naive log design and 4 times the number of pages with advanced log design, indicating that log update is not a suitable lifetime solution for LISA. Instead of using log update, LISA can only update values using RMW operations (step ❸) so as able to update new page addresses into local model (step ❹). In this case, all two flash pages shall be rewritten to follow the neighboring-keys clustering constraint.

This problem grows increasingly serious when page utilization drops along with the LISA's execution. Initially, the utilization of most flash pages in the SSD is nearly 100% right after the LISA's initialization, but it drops substantially after LISA serves a series of key updates. Compared with the high average page utilization, the SSD requires accessing more pages while running value updates or range queries[1], when the SSD suffers from low average page utilization. Writing data to more flash pages hurts SSD's lifetime and reading more pages from the SSD increases the response time for the query. To validate the observations, we build an emulator by integrating LISA with SSD and run the experiments on a dataset, containing ten million of two-dimensional key-value data. We demonstrate the evaluation results in Figure 4, on evaluating page utilization and the lifetime and performance impacts on SSDs. Specifically, to evaluate the lifetime impact, our emulator randomly updates one million key-value data and monitors the total number of page writes.

To evaluate the performance impact, our emulator executes ten thousand of range queries and keeps track of the total pages read from the SSD. Figure 4(a) shows the profiling result of SSD's page utilization across a series of key updates. The x-axis shows the percentage of updated data inside the dataset, and the y-axis demonstrates the percentage of utilization level (i.e., high, middle, or low utilization) among all pages. It's evident that even updating a few key-value data significantly impacts page utilization. This occurs because LISA divides high-utilization pages into several low-utilization pages to accommodate newly inserted data, preventing newly allocated pages from falling below 50% utilization.

[1]Range queries work by searching all key-value data within a given range.



(a) Page Utilization. (b) Number of Page Accesses.

Figure 4: Page Utilization Analysis & Impacts on SSDs

Figure 4(b) shows how the page utilization impacts lifetime and performance during executing value updates and range queries, respectively. Generally, data in key-value stores updates over time; thus, the average page utilization across all flash pages fluctuates. We snapshot our emulator (including LISA's local model and data layout on the SSD) when the average page utilization reaches a predefined value and run value updates or range queries separately. The x-axis shows four average page utilization, and the y-axis demonstrates the accessed pages normalized to the result while the average page utilization is 99%. LISA requires writing 1.14 times more pages during serving value updating and reading 1.55 times more pages during executing range queries when the page utilization drops from an initial 99% to 68%. Retraining LISA can enhance page utilization; however, this process requires data reorganization, leading to a significant increase in write traffic inside the SSD, thereby worsening the lifetime issue. Also, during the data reorganization, LISA must be paused, consequently impacting the service response time. Furthermore, as depicted in Figure 4(a), merely updating a few key-value data can substantially reduce page utilization, thereby diminishing the efficacy of the retraining process.

This work is strongly motivated by the need to enhance SSD lifetime and accelerate range query performance when running LISA on SSDs. We propose a middleware to extend LISA to adaptively manage spatial key-value data with considering the SSD's update constraint. Our ultimate goal is to minimize running extra page operations while serving value updates or range queries by keeping high page utilization. The major technical challenge falls on how to reallocate and compact updated key-value data into fewer pages while adhering to LISA's neighboring-keys clustering constraint and SSD's out-of-place update constraint.

3 ALISA: ADAPTIVE LEARNED INDEX STRUCTURE FOR SPATIAL DATA

This section presents our Adaptive Learned Index structure for Spatial dAta on SSDs (ALISA) to persistently monitor updated data distribution and adaptively restructure to align

with SSD access characteristics. Instead of flushing all pages accommodating updated key-value data, ALISA smartly reallocates and compacts updated key-value data into fewer pages via updating local models. The rationale behind this design is that local models can be adjusted at the runtime as long as they comply with the neighboring-keys clustering constraint.

Technically, we design a middleware between LISA and the SSD, and no modifications are needed for both LISA and the FTL design inside the SSD. Section 3.1 introduces the adaptive update-interval model expander to minimize the number of written pages. Section 3.2 presents the adaptive model compactor to minimize the number of written pages while also making pages as compact as possible. Section 3.3 then introduces the model-aware data pinner to reduce unnecessary writes by pinning data of the least updated pages in the write buffer. For simplicity, unprocessed updated pages are termed "unresolved pages"; once processed by our methods, we call them "resolved pages".

3.1 Adaptive Update-Interval Model Expander

We propose an adaptive update-interval model expander to reduce the amount of page writes on SSDs by only flushing those must-to-flush data from the write buffer instead of naively flushing all pages accommodating updated data. To conform to the neighboring-keys clustering constraint after each flushing, the must-to-flush data includes updated data and non-updated data sitting between any two updated data. To identify those must-to-flush data, our design maintains an "update interval" which denotes the range spanning from the minimum to the maximum mapped values of updated data across consecutive pages. Technically, our model expander maintains the update interval for consecutive unresolved pages and checks whether the data within this interval can fit on one page. If it fits, our model expander reallocates these data to a new page, marks it as invalid on the original pages, and inserts the new page address into the local model.

We illustrate a toy example to explain our model expander in Figure 5, assuming $m_{A1}<m_{A2}<m_{A3}$ and $m_{A4}<m_{A5}<m_{A6}$, the red color denotes updated data and the update interval extends from m_{A2} to m_{A4}. Since there are only three data points within the update interval, our model expander reallocates them onto a new page, thus decreasing the number of pages written from two to one. The reallocation does not violate the neighboring-keys clustering constraint because it utilizes the "update interval" concept, which moves the entire data range rather than relocating scattered updated data onto a new page. Nevertheless, the model expander results in a substantial drop in space utilization because it requires additional pages for the same data volume. Considering this, we develop an adaptive model compactor to address this issue.

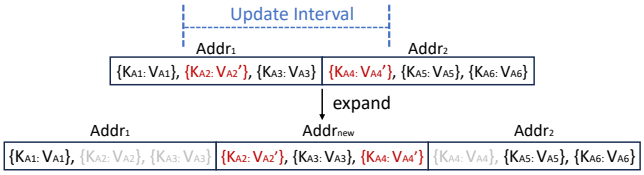


Figure 5: Adaptive Update-Interval Model Expander

3.2 Adaptive Model Compactor

Solely using the model expander can lead to a significant decrease in space utilization. We propose an adaptive model compactor to increase space utilization while also reducing the number of pages written. The model compactor assesses whether data on n consecutive unresolved pages can fit into $n-1$ pages. If feasible, the model compactor consolidates this data onto fewer new pages, invalidates the original pages, and modifies the local model. Here, n can be any number not greater than the length of local model. We limit n to a maximum of three, as the model expander typically generates three partially filled pages, and a higher n value leads to diminished gains in space utilization and extended processing duration.

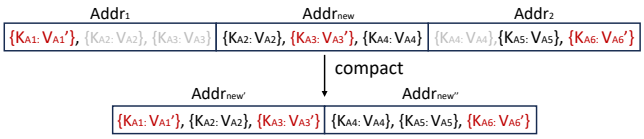


Figure 6: Adaptive Model Compactor

As shown in Figure 6, we continue from the outcome of the previous iteration of the model expander, with data in all three pages being updated this time. Since there are only six data points on these two pages, our model compactor consolidates them onto two new pages, thus decreasing the number of pages written from three to two. Our model compactor not only fixes the space issues caused by the model expander, but it also has a chance to improve LISA's original low page utilization. This improvement happens because the expander and compactor work together to rearrange how data is stored in a shard, potentially enhancing previous poor utilization. More details of the experimental results are provided in Section 4.2.

3.3 Model-aware Data Pinner

Following processing through the model expander and model compactor, there might still be unresolved pages remaining in the buffer. We propose a model-aware data pinner to avoid naively flushing these unresolved pages into the SSD. The data pinner allocates a portion of buffer to pin[2] or (lock) appropriate data in DRAM. We expect these data points will either be processed by the model expander and model compactor in the future, or written into the SSD along with more upcoming updated data as unresolved pages.

Technically, our data pinner maintains a model-aware min-heap to track the number of updated data points on unresolved pages. Each node in the min-heap contains two elements, the identification of the unresolved page and the number of corresponding updated data points in the write buffer. The number of updated data points belonging to this unresolved page serves as the key of the min-heap structure. Once the write buffer is determined to flush into the SSD, the model-aware data pinner will select and pin updated data of the least updated unresolved pages to populate the pinned area. We exhibit an example to illustrate the model-aware pin structure in Figure 7. Our data pinner will select updated K-V items belonging to the top-n unresolved pages in the pin structure (P_c in this example). Subsequently, our data pinner invokes the kernel function to pin K-V items (i.e.,$\{K_{c1}$: $V'_{c1}\}$) and any other non-pinned K-V items will be flushed into the SSD. Notably, pinning too many items reduces the available space for buffering newly updated key-value items, thereby increasing the middleware's management time. Based on our testing, we suggest limiting the overall pinning size to no more than 5% of the overall write buffer size.

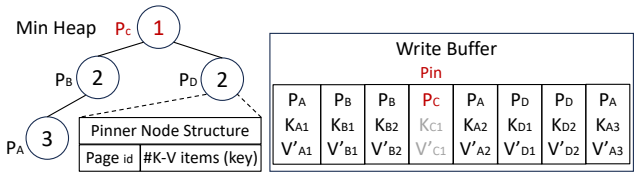


Figure 7: Model-aware Data Pinner

Algorithm 1 describes how ALISA intelligently reallocates and compacts updated key-value data with its three components. When the write buffer is full and determined to flush, ALISA categorizes updated data based on their shard IDs and retrieves corresponding local models for further modifications. ALISA handles the updated data on a shard-by-shard basis (*Line* 1). Initially, ALISA triggers the model compactor to concurrently reduce the number of written pages and improve space utilization. The model compactor gathers consecutive unresolved pages and assesses if they can be condensed into fewer pages (*Line* 4). If feasible, it consolidates them into new pages, invalidates the original pages, and adjusts the local model accordingly (*Line* 5). For the remaining updated data, ALISA initiates the model expander to reduce the written pages. The model expander will collect consecutive unresolved pages, calculate the update interval, and assess whether the data within this interval can fit on one page (*Line* 8). If feasible, the model expander reallocates these data to a new page, marks it as invalid on the original pages, and inserts the new page address into the local model (*Line* 9). Finally, ALISA allocates a portion of the write buffer as a pinned area and utilizes the min-heap to pin selected updated data in the write buffer (*Line* 11, 12).

[2]Modern Linux kernel provides pin_user_pages() and unpin_user_pages() function to pin and unpin user pages in the memory [3].

Algorithm 1: ALISA Update Processing

```
   Input: LM = {local model lm | lm.updated_cnt() > 0 }
1  for lm ∈ LM do
       /* 1. Model Compactor                          */
2      for page P ∈ lm.PA do
3          Ps = P and consecutive pages;
4          if Ps.unresolved() & Ps.compactible() then
5              lm.compact(Ps);
       /* 2. Model Expander                           */
6      for page P ∈ lm.PA do
7          Ps = P and consecutive pages;
8          if Ps.unresolved() & Ps.expandible() then
9              lm.expand(Ps);
   /* 3. Data Pinner                                  */
10 min_heap = heapify(unresolved pages);
11 while pinned area is not full do
12     pop(min_heap) and pin updated data;
```

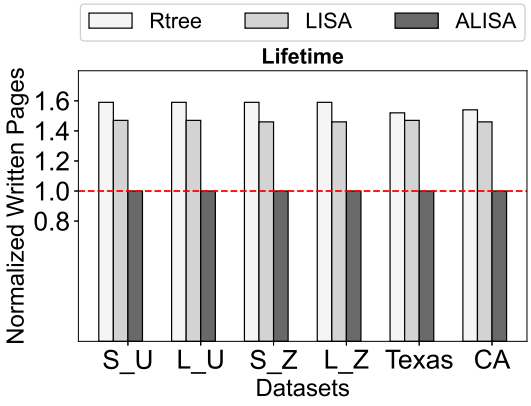
4 PERFORMANCE EVALUATION

4.1 Performance Metrics and Evaluation Setup

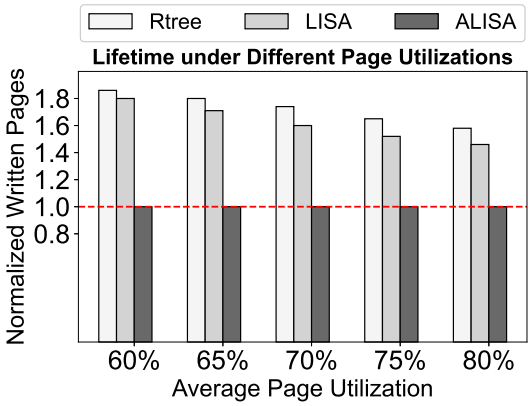
To validate the effectiveness of our design, we build an emulator by integrating ALISA with SSD, where ALISA is implemented by merging LISA with our middleware. We evaluate the effectiveness of the proposed middleware for improving the lifetime of SSDs and page utilization of spatial database during value updates, which reduces the I/O costs of subsequent range queries. In particular, we evaluate above metrics by comparing the proposed ALISA with two baseline approaches, i.e., original LISA and the most popular spatial index R-tree. To obey the neighboring-keys clustering constraint, LISA can only use RMW operations when

updating data. Given that most computer systems employ write buffers to aggregate and optimize write operations, a set of value-update operations is buffered in DRAM and then batch-written to SSDs for all approaches.

There are four synthetic datasets and two real world datasets in our experiment. We developed four synthetic datasets, formed by a combination of two types of distributions - uniform and Zipf [32] - and two sizes – small (S) and large (L) containing twenty million and sixty million two-dimensional key-value items respectively. We collected two real world datasets - Texas and California building footprints released by Microsoft, each containing 10,677,005 and 11,528,930 multipolygons respectively. We simplified the multipolygons to point type by calculating the means of latitude and longitude. To simulate system usage over time, for each dataset, we randomly select half of data to form the initial dataset I for building LISA and R-tree. The remaining half of data constitutes dataset E to be inserted into the databases. From the union of I and E, we randomly select half of data to be deleted. A portion of memory is configured as a write buffer. In our experiment, we set 5% of the size of each dataset as the write buffer size. After initialization, our emulator uniformly selects and updates the values of the whole dataset with the configured write buffer and logs the total number of pages written. After updating, we issue and process the same set of ten thousand range queries using all approaches. Each side of every query rectangle is assigned a random length, ranging from zero to 25% of the corresponding axis's length. In our testbed, we run the emulator on the machine with Intel Core i7-12700 CPU and 128GB of DRAM. The operating system is Ubuntu 22.04.1 and the Linux kernel is 6.2.0-33-generic. We use"FIO" (version 3.28) to send requests to the SSD for measuring the read time and write time.



(a) Evaluation on Lifetime



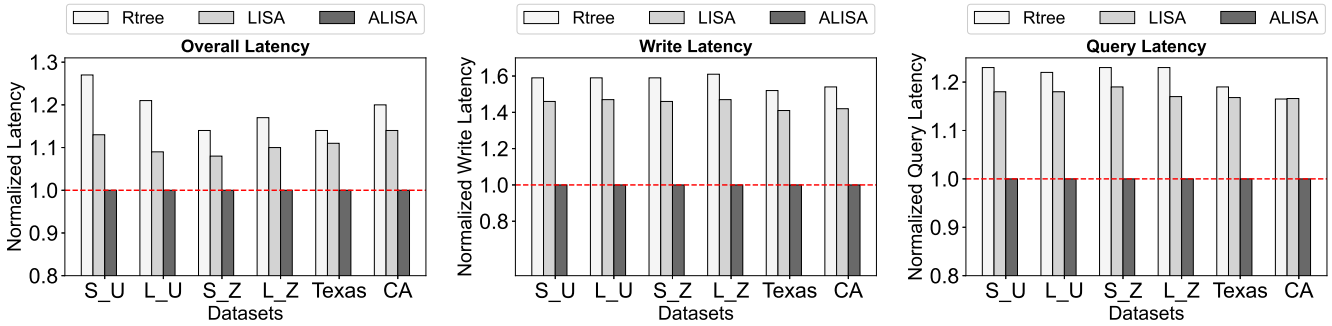
(b) Evaluation on Lifetime under Different Utilizations

Figure 8: Evaluation on Lifetime and Detailed Lifetime Analysis under Different Page Utilizations.

4.2 Evaluation Results

4.2.1 Evaluation on Lifetime and Analysis under Different Page Utilizations. Figure 8 provides the results of the SSD lifetime and detailed lifetime analysis with different page utilizations, under running ALISA against LISA and R-tree. Figure 8(a) displays the number of written pages during value updates, where the x-axis indicates six datasets and the y-axis represents the normalized written pages compared to ALISA during these updates. The results show that ALISA can save around 32% pages written to the SSD relative to LISA and 37% pages written compared to R-tree. The reason is that ALISA continuously monitors the data distribution in the write buffer and minimizes the pages written with the model expander and the model compactor. That is, ALISA determines if the key-value items within the update interval can be merged onto a single page, or if multiple unresolved pages can be condensed into fewer pages to reduce the number of written pages. Furthermore, the model-aware data pinner contributes to pin the least updated unresolved pages in the write buffer, aiming to boost the frequency of such instances in the future.

Figure 8(b) provides the results of the SSD lifetime analysis under different page utilizations. During the experiment databases are constructed, the average page utilization across all flash pages fluctuates. We snapshot our emulator when the average page utilization reaches a predefined value and update the values of the entire dataset to observe the lifetime improvement. To be specific, we take small size uniform dataset and insert data of dataset E into initial dataset I to reach the certain page utilization. Figure 8(b) displays the number of written pages during value updates, where the x-axis indicates different page utilizations and the y-axis represents the normalized written pages compared to ALISA during these updates. The results show that ALISA can save 32%~44% pages written to the SSD relative to LISA and



(a) Evaluation on Overall Latency (b) Evaluation on Write Latency (c) Evaluation on Query Latency

Figure 9: Evaluation on Latency.

37%~46% pages written compared to R-tree. The findings indicate that ALISA consistently improves lifespan across different levels of page utilization. Specifically, it thrives in scenarios of low page utilization, where the activation of the model expander and model compactor becomes more probable.

4.2.2 Evaluation on Latency. We then show a detailed performance evaluation in Figure 9. Figure 9(a) demonstrates the overall latency for mixing value updates operations and further ten thousand range queries, under running ALISA against LISA and R-tree. The method overhead is also measured and included in ALISA overall running time. The x-axis indicates six datasets and the y-axis represents the normalized latency compared to ALISA. The results show that R-tree exhibits 14%~27% longer running time and LISA exhibits 8%~14% longer running time compared to ALISA. Figure 9(b) displays the write latency for value updates operations, where the x-axis indicates six datasets and the y-axis represents the normalized write latency compared to ALISA. The results show that R-tree exhibits up to 60% longer write time and LISA exhibits up to 47% longer write time compared to ALISA, attributed to ALISA's efficiency in reducing the total number of written pages. Figure 9(c) shows the query latency for ten thousand range queries, where the x-axis indicates six datasets and the y-axis represents the normalized query latency compared to ALISA. The results show that R-tree exhibits up to 23% longer query time and LISA exhibits up to 19% longer query time compared to ALISA. The results imply that running ALISA can enhance page utilization so as to reduce accessed pages during range queries. In fact, the average page utilization of LISA and R-tree remains at approximately 68% and 65% after value updates, while ALISA's average page utilization improves from 68% to around 80% thanks to the model compactor. This result proves that the system can achieve good space utilization and efficiency without the need for retraining the models, which seriously delays the response time and wears out SSDs cells earlier.

5 CONCLUSION

Aiming to enhance SSD lifetime and accelerate range query performance when running LISA on SSDs, we propose a middleware to extend LISA to ALISA to adaptively manage spatial key-value data with considering the SSD's update constraint. Our middleware comprises three main components: an adaptive update-interval model expander to minimize the number of written pages by keeping track of update intervals, an adaptive model compactor to minimize the number of written pages while also making pages as compact as possible, and a model-aware data pinner to reduce unnecessary writes by pinning data of the least updated pages in the write buffer. The evaluation demonstrates that ALISA effectively enhances the lifetime of SSDs by reducing 32%~44% and 37%~46% of pages written to the SSD compared to LISA and R-tree, respectively. Moreover, ALISA achieves faster execution times, completing range queries approximately 15% and 19% faster than LISA and R-tree, respectively, attributed to improved page utilization. Specifically, the average page utilization of LISA and R-tree remains at approximately 68% and 65% after value updates, while ALISA's average page utilization improves from 68% to around 80%.

6 ACKNOWLEDGEMENT

This work was supported in part by National Science and Technology Council under grant nos. 113-2628-E-A49-021, and 112-2222-E-A49-002-MY2 and Ministry of Education under Yushan Young Fellow Program. (Corresponding Author: Chun-Feng Wu)

REFERENCES

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points

and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. {GAIA}: An {OS} page cache for heterogeneous systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 661–674, 2019.

[4] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. A generic and efficient framework for spatial indexing on flash-based solid state drives. In *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September, 2017, Proceedings 21*, pages 229–243. Springer, 2017.

[5] Liang Chen, Sarang Thombre, Kimmo Järvinen, Elena Simona Lohan, Anette Alén-Savikko, Helena Leppäkoski, M Zahidul H Bhuiyan, Shakila Bu-Pasha, Giorgia Nunzia Ferrara, Salomon Honkala, et al. Robustness, security and privacy in location-based services for future iot: A survey. *Ieee Access*, 5:8956–8977, 2017.

[6] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[8] Harish Doraiswamy and Juliana Freire. A gpu-friendly geometric data model and algebra for spatial queries. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1875–1885, 2020.

[9] Paul DuBois. *MySQL*. Addison-Wesley, 2013.

[10] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[12] Haosheng Huang, Georg Gartner, Jukka M Krisp, Martin Raubal, and Nico Van de Weghe. Location based services: ongoing evolution and research agenda. *Journal of Location Based Services*, 12(2):63–93, 2018.

[13] Peiquan Jin, Xike Xie, Na Wang, and Lihua Yue. Optimizing r-tree for flash memory. *Expert Systems with Applications*, 42(10):4676–4686, 2015.

[14] Scarlett T Jin, Hui Kong, and Daniel Z Sui. Uber, public transit, and urban transportation equity: a case study in new york city. *The Professional Geographer*, 71(2):315–330, 2019.

[15] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

[16] Myoungsoo Jung and Mahmut T Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535. IEEE, 2014.

[17] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504, 2018.

[18] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 2119–2133, 2020.

[19] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[20] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. R-trees have grown everywhere. Technical report, Technical Report available at http://www. rtreeportal. org, 2003.

[21] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2015.

[22] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[23] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 985–1000, 2020.

[24] Maciej Pawlik and Wojciech Macyna. Implementation of the aggregated r-tree over flash memory. In *Database Systems for Advanced Applications: 17th International Conference, DASFAA 2012, International Workshops: FlashDB, ITEMS, SNSM, SIM 3, DQDI, Busan, South Korea, April 15-19, 2012. Proceedings 17*, pages 65–72. Springer, 2012.

[25] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. Gpu-based parallel r-tree construction and querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627. IEEE, 2015.

[26] Grace Samson, Joan Lu, and Qiang Xu. Large spatial datasets: Present challenges, future opportunities. In *Proceedings of the International Conference on Change, Innovation, Informatics and Disruptive Technology ICCIIDT'16, London-UK, October, 2016*, pages 204–217, 2016.

[27] Camélia Slimani, Chun-Feng Wu, Stéphane Rubini, Yuan-Hao Chang, and Jalil Boukhobza. Accelerating random forest on memory-constrained devices through data storage optimization. *IEEE Transactions on Computers*, 2022.

[28] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pages 569–574. IEEE, 2019.

[29] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient r-tree implementation over flash-memory storage systems. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pages 17–24, 2003.

[30] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. Joint management of cpu and nvdimm for breaking down the great memory wall. *IEEE Transactions on Computers*, 2020.

[31] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. When storage response time catches up with overall context switch overhead, what is next? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4266–4277, 2020.

[32] Chun-Feng Wu, Martin Kuo, Ming-Chang Yang, and Yuan-Hao Chang. Performance enhancement of smr-based deduplication systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(9):2835–2848, 2021.

[33] Chun-Feng Wu, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. A joint management middleware to improve training performance of deep recommendation systems with ssds. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 157–162, 2022.

[34] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. {LSM-trie}: An {LSM-tree-based}{Ultra-Large}{Key-Value} store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, 2015.