SWIX: A Memory-efficient Sliding Window Learned Index

LIANG LIANG* and GUANG YANG*, Imperial College London, UK
ALI HADIAN, Imperial College London, UK
LUIS ALBERTO CROQUEVIELLE, Imperial College London, UK
THOMAS HEINIS, Imperial College London, UK

Data stream processing systems enable querying over sliding windows of streams of data. Efficient index structures for the streaming window are a crucial building block to enable querying the sliding window for operations such as aggregation and joins. This paper proposes SWIX, a novel memory-efficient learned index for sliding windows. Unlike conventional learned indexes that rely on tree structures to achieve logarithmic query cost, SWIX has a flat structure that uses substantially less memory and enables efficient query execution while having a low cost for index maintenance when inserting (and retraining). SWIX dynamically adapts itself to the real-time distribution shifts of data streams.

SWIX outperforms existing indexes in terms of query execution time and memory footprint for workloads characterized by very frequent updates. Our results show that SWIX has a significantly smaller memory footprint than conventional, streaming, and learned indexes, using only 22% to 42% of the size compared to state-of-the-art approaches, yet outperforming them by up 1.2× to 1.6× on average (and up to 52×) in terms of query time, making it a space- and time-efficient method for indexing data streams. For concurrent learned indexes, Parallel SWIX can achieve up to 3.45× throughput with only 34% of memory consumption.

CCS Concepts: • **Information systems** → **Data structures**.

Additional Key Words and Phrases: learned index, flat structure, lightweight structure, parallel structure, streaming processing, window-based query, automatic tuning, update heavy

ACM Reference Format:
Liang Liang, Guang Yang, Ali Hadian, Luis Alberto Croquevielle, and Thomas Heinis. 2024. SWIX: A Memory-efficient Sliding Window Learned Index. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 41 (February 2024), 26 pages. https://doi.org/10.1145/3639296

1 INTRODUCTION

Processing data streams has become increasingly important as large-scale online services continue to grow [1, 37]. In a streaming environment, systems face significant challenges due to the amount of data and the speed at which it arrives. Large-scale stream processing systems thus require efficient data structures and algorithms to be able to process data in real time and make timely decisions.

Streaming systems manage a finite amount of the most recent data in a sliding window. This works like a queue that captures new records as they arrive and removes older ones to avoid overflowing memory. For example, a network traffic monitoring system can use a sliding window

*Both authors contributed equally to this research.

Authors' addresses: Liang Liang, liang.liang20@imperial.ac.uk; Guang Yang, guang.yang15@imperial.ac.uk, Imperial College London, London, UK; Ali Hadian, ali.hadian@gmail.com, Imperial College London, UK; Luis Alberto Croquevielle, a.croquevielle22@imperial.ac.uk, Imperial College London, UK; Thomas Heinis, t.heinis@imperial.ac.uk, Imperial College London, UK.

ACM 2836-6573/2024/2-ART41
https://doi.org/10.1145/3639296

to keep track of recent traffic data for real-time monitoring and analysis. Stream processing systems need to be able to efficiently search through the records in a sliding window, such as by IP address or packet ID.

Managing a sliding window is challenging as it requires handling frequent updates and efficient search through the records. In a non-streaming system, records are inserted, deleted, and searched based on the primary key. However, in a window-based stream processing system, data arrives and expires based on timestamps and is searched using the primary key, which is typically a unique identifier, such as a record ID. Therefore, streaming data structures must be able to handle frequent updates based on timestamps and search efficiently over records.

Several indexes have been suggested for Index-based window processing (IBWP), particularly for streaming joins [35]. Their design uses multiple data structures optimized for read and update performance. The drawback of these designs is the extensive memory overhead from storing multiple data structures, such as a large and a small B-tree, to handle the sliding window. An alternative approach is to use learned indexes, which exploit patterns in data distribution to predict the location of the records [23]. While some learned indexes [9, 11, 45] support updates and achieve speedup over conventional indexes, they do not achieve the same level of compactness compared to conventional indexes. These existing learned indexes cannot handle streaming updates, where updates are based on a non-indexed time dimension, without external structures or extensive scans. FLIRT [44], the only streaming learned index, is designed to update and query solely based on the time dimension. This requires the primary keys to be time-ordered.

In this paper, we develop a novel memory-efficient updatable learned index for IBWP in streaming workloads. Our main design goal is to create substantially more compact data structures than traditional or learned indexes while providing similar or better query performance compared to existing approaches. To achieve this, our design is based on a flat two-layer data structure that reduces the height of the index compared to tree-based structures, thereby reducing redundancy in the index. The flat structure also allows the index to use parallelism to enhance performance. To handle updates efficiently, we dynamically scale the data structure according to the data distribution to achieve fast read performance while minimizing the memory overhead. Additionally, we implicitly delete old data from the index while searching and updating the index to ensure compactness.

The result is **SWIX**, the **S**liding **W**indow Learned **I**nde**X**, a two-layer lightweight learned index that can efficiently manage IBWP workloads under very high update rates, commonly found in streaming applications, and supports parallelism. The flat structure ensures compactness while reducing traversal and rebalancing time costs in IBWP. SWIX uses a novel underlying data structure SWarray, combining the location prediction of learned indexes with dynamic restructuring mechanisms to compact the index during updates. Specifically, SWarray expands when the index ingests new data from the stream and shrinks when discarding old data. The expansion mechanism generates space for insertions, while the shrinkage mechanism ensures the compactness of the array. An online auto-tuning algorithm controls the expansion and shrinkage to optimize the array under changing data distributions and query workloads without additional space overheads. SWIX uses a bulk-load algorithm to cold-start the index while the optimal configuration is still unknown to the auto-tuner. Finally, we introduce Parallel SWIX, which supports parallel reads and updates through data partitioning.

The remainder of this paper is organized as follows. We motivate our work in Section 2. The design choices of SWIX are in Section 3, followed by an explanation of the operations in Section 4. We then present Parallel SWIX in Section 5. An experimental evaluation shows results and analysis in Section 6. Finally, we present the related work in Section 7 and conclude in Section 8.

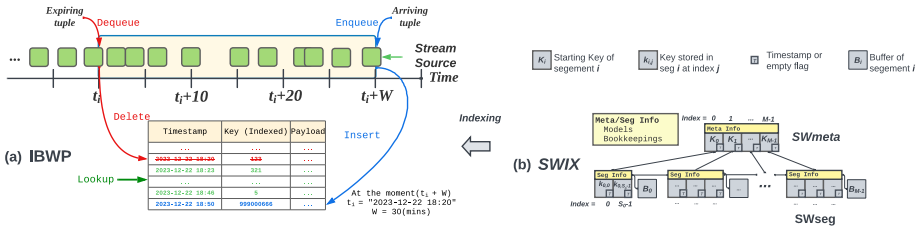


Fig. 1. (a) IBWP and (b) SWIX Structure.

2 MOTIVATION

2.1 Index-Based Window Processing (IBWP)

Figure 1a illustrates a sliding window and different operations in IBWP. The system receives records from a stream and stores the most recent ones in the sliding window. Due to memory constraints, an IBWP system keeps the latest records in the past W time units (e.g., seconds) from the stream, where W is the window frame. Alternatively, the window can be defined with a fixed number of tuples, commonly known as the tuple-based window. Although they have different primary characteristics, both paradigms use timestamps/tuples to ensure data validity.

The sliding window is a queue, where new records are enqueued to the window and older records are dequeued and deleted. The system also supports looking up records in the window using a primary key. This problem setting generalizes the approach in FLIRT [44], where keys are sequential and data is deleted from the front and inserted at the back. In contrast, SWIX aims to handle a generalized IBWP workload, where insertions and deletions occur anywhere within the index (where the index is key-sorted and not time-sorted).

Existing methods for indexing IBWP workloads are either inefficient in terms of memory usage or update/search performance. Current streaming indexes are agnostic to the data distribution and hence do not exploit it to reduce the index size. Equally, learned indexes that exploit the data distribution to reduce index size are not efficient in supporting updates in IBWP workloads (enqueue and dequeue) which are frequently performed as the window slides over the data stream. To support updates, existing updatable learned indexes include many redundancies in their design, so they have to sacrifice space to ensure efficient queries.

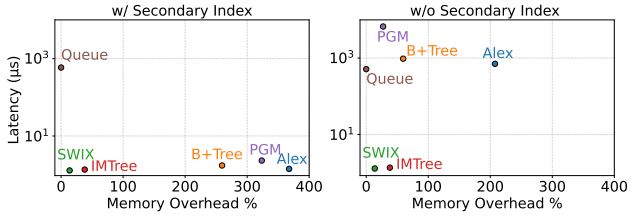


Fig. 2. Design tradeoff for indexing streaming data.

2.2 The Trade-off Between Latency and Memory

Figure 2 illustrates the trade-off between latency and memory overhead of existing methods and SWIX. We compare the state-of-the-art (with and without secondary indexes) and SWIX, considering the lookup time and memory overhead of each index. FLIRT [44] is not included as it does not support random updates. The latency consists of the time for enqueue, dequeue, and search, with equal ratios for each operation.

Indexes that are not designed for data streams, such as the B+tree and learned indexes (such as PGM [11] and ALEX[9]), require a secondary index to achieve low latency for IBWP workloads. The secondary index provides a timestamp-to-key mapping to identify which keys to delete from the primary index. Without a secondary index, these indexes have high latency from scanning the index for the expired records (Figure 2-right). However, using a secondary index results in a large combined memory overhead of about 3× the data size (Figure 2-left). Therefore, traditional and learned indexes are not efficient for streaming workloads, as they are either inefficient in terms of memory consumption or query execution time.

Queues are very lightweight and efficient for enqueue and dequeue operations. Note that the queue is time-sorted, hence, search operations require linear searching through all the records as the primary keys are unsorted. In this regard, SWIX aims to provide an efficient trade-off between memory consumption and speed. We consider IMTree [35], a streaming index, to be the main rival for our method. However, IMTree is based on B+Trees, which results in a much higher memory overhead compared to SWIX.

3 SWIX DESIGN

SWIX is a space- and time-efficient learned index for update-heavy streaming workloads. SWIX takes advantage of several key insights, including a two-layer structure, a piecewise linear learned model, a configurable mechanism for optimizing search and update performance, an online auto-tuner for adapting to changing distributions, and a parallel framework for scalability.

3.1 Design considerations

To create a space- and time-efficient updatable learned index for streaming settings, we explore using a flat structure to reduce index size. This is in contrast to most updatable learned indexes [9, 11, 13, 26, 35, 40] as well as B+Trees, which use tree structures. They are inefficient for streaming workloads due to the redundancy in hierarchical tree structures. For example, in a B+Tree, the number of redundant keys is $n/(2d)$, where n is the data size and $2d$ is the branching factor. As the branching factor of a tree structure increases, the tree becomes shallower, and the memory overhead diminishes. This is illustrative of all indexes using a tree structure, and not only the case for B+Trees. For SWIX, we use a flat structure with only two levels. As mentioned, this reduces the memory overhead by lowering the number of redundant keys. In Section 9, we show theoretical evidence that two levels are sufficient to get a significant speed-up over traditional indexes like B+Tree. Moreover, adding more levels does not necessarily result in lower search times, especially if the data distribution is hard to learn. Hence, we prioritize the memory advantages of a flat structure over the uncertain gains in latency of adding more levels. While FLIRT [44] also uses a two-level design, it only handles changes in the front and back of the index as it assumes the key equals the timestamp. On the other hand, SWIX handles data distribution shifts appearing in different parts of the index simultaneously and is more general-purposed.

Designing SWIX to have fewer levels means increasing the number of elements in each node, which makes it challenging to efficiently search through each node. To address this, we use piecewise linear functions, which can give low prediction error and thus allow for time-efficient lookup over a space-efficient flat data structure. Piecewise linear functions are also lightweight to retrain, and the prediction error is controlled by the number of functions (segments).

An additional challenge for managing a sliding window is that updates in the data stream can reduce prediction accuracy due to shifts, known as *drifts* in data distribution. Therefore, periodic retraining is necessary to ensure the accuracy of the model. In SWIX, we reduce the impact of drifts by using more space. While ALEX [9] has a slightly similar approach of smoothing out the data distribution by reserving gaps between data (preventing the predictive model from mapping

adjacent inserted keys to the same position). We believe that using a static number of gaps results in undue space overhead. Instead, SWIX uses the data distribution to dynamically generate (or remove) gaps on an ad-hoc basis.

We implement the dynamic mechanism with an online auto-tuner based on the cost model that monitors the data distribution and self-configures the index to be memory efficient while guaranteeing the speedup comparable to existing learned indexes.

To further improve the performance of SWIX, we parallelize SWIX into distinct partitions to support parallel reading, writing, and retraining. Unlike current scalable learned indexes [26, 40], which are based on a concurrent design, SWIX reduces the workload in each core and allows for different parts of SWIX to update and retrain simultaneously without locking.

3.2 SWIX Structure

The structure of SWIX is shown in Figure 1b. SWIX uses SWarray, a novel base structure that combines a linear prediction model for efficient querying and a dynamic expanding/shrinking mechanism (see Section 4) for efficient updates while remaining lightweight and accurate. SWarray uses a continuous memory layout to ensure data locality and scan performance. Furthermore, SWarray implicitly discards expired data without requiring a secondary index. We optimize SWarray for common workloads on each level:

3.2.1 SWseg. SWseg is optimized for handling keys on the lower layer of the index. In terms of notation used in tree structures, SWseg can be thought of as a leaf node. As SWsegs are frequently updated, we expand SWarray when needed to generate gaps for insertion. The number of gaps depends on the current data distribution and is controlled by the auto-tuner. We use the existing trained model to allocate gaps based on where the model predicts empty spaces. If neighboring keys are closer together (in terms of range), they are predicted closer to each other. Hence, the potential for inserting other keys between the neighboring keys is lower, which results in fewer gaps. Conversely, if the two keys are further away from each other, the potential for intermediate keys is much higher. We use the trained model to compact and encapsulate these relationships to save training time. Furthermore, we argue that it is difficult to predict the position of gaps when the distribution shifts. Rather than increasing the number of gaps (which increases space overhead), SWIX opts to reduce the number of gaps and uses a cache-efficient buffer as "universal" gaps to absorb updates in dense areas. We implement the buffer using a simple sorted array with a fixed length, which can be updated with any standard array procedures. The buffer also absorbs the accuracy loss from drifts by storing inserted keys without increasing the error bound. We limit the buffer size and compact any expired key from the buffer to ensure memory efficiency. Neighboring pointers connect segments together to improve range scan performance.

3.2.2 SWmeta. SWmeta is optimized for the upper level, and its main task is to manage the SWsegs (corresponds to the root node in a tree structure). The meta-node stores the starting key of each segment. Therefore, the entire meta-node consists of redundant keys to improve the time performance. Updates in SWmeta are infrequent and only occur when one or more segments are retrained. Hence, the main objective of the meta-node is to ensure compactness while maintaining good search performance to locate corresponding SWsegs. For this reason, we can avoid the use of a buffer. SWmeta stores a bitmap[1] to all segments to allow them to retrain together as a form of merge mechanism to prevent segments from fragmenting.

[1]The bitmap is a compact representation of an array of booleans, storing each boolean as a bit. Implementation-wise, our bitmap uses a `C++` vector of 64-bit unsigned integers. The size of the bitmap is `ceil`(number of items/64).

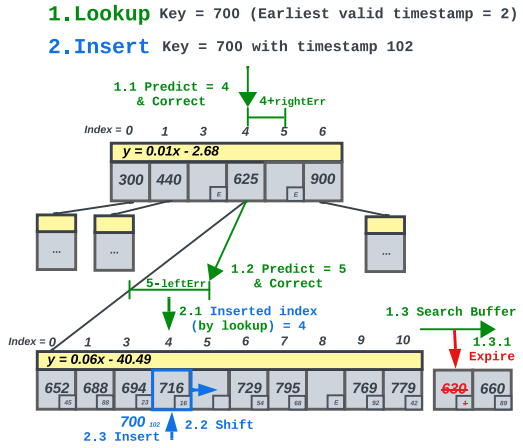


Fig. 3. Example of search and insertion. The `lookup` for key 700 employs two *predict-correct* steps to find its segment layer position. However, because the key isn't present in the segment, the buffer is searched. Simultaneously, the invalid data (key 630) is implicitly `expired`. To `insert` the key, we determine its position in the segment layer and insert the data according to the current state of the segment.

4 SWIX OPERATIONS

4.1 Search

The core procedure to look up a key is a *predict-correct* process: (1) predicting the position and (2) correcting the error. This procedure is first performed in SWmeta to identify the segment that may store the key before proceeding to locate the key with the same procedure in SWseg. The buffer is searched (using binary search) if no key is found in the segment's SWarray. For range queries, we start with a point lookup for the range's lower bound. Once we reach the segment level, we scan through the segments via neighbor pointers until we reach the range's upper bound. An example is illustrated in Figure 3.

We optimize the correction step to use bounded exponential search, although any "last-mile search" algorithm can be used. Search bounds ensure the worst-case search performance of the correction step and give SWIX an analytical search cost. To ensure search performance when the search bounds expand with insertions, SWIX opts to use exponential search over binary search as insertions are prioritized to be placed close to their predicted positions (details in Sections 4.2.1).

We optimize for streaming updates by erasing invalid data during searches and scans (highlighted in red in Figure 3). This eliminates the cost of explicit deletion and the need for any external structures. Once invalid data (e.g., expired keys) are found, we convert them to gaps in SWarray and erase them in the buffer. To ensure memory efficiency, we compact the segment (via retraining) when they reach the maximum number of gaps unique to each segment (50% of the initial segment size). If the entire segment consists of invalid data, SWmeta flags the segment as a gap.

4.2 Updates

Updates come in the form of insertions (enqueue) and deletions (dequeue). As updates mainly affect SWseg, we dynamically change the size of the SWseg and the number of gaps according to the data distribution. To reduce prediction errors from drifts, we limit the error in SWseg and use the buffer to absorb error caused by distribution change. The effect of updates has a lagging effect on SWmeta as the segments naturally buffer the changes.

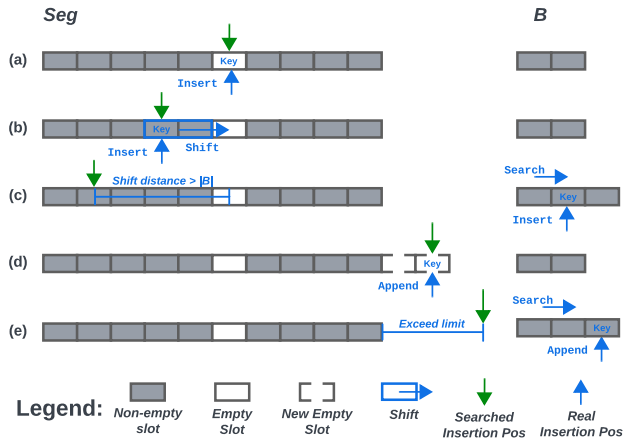


Fig. 4. Insertion strategies for each insertion situation.

4.2.1 Insertion. The insertion procedure starts with a lookup to find the insertion position. The procedure is shown in blue in Figure 3, followed by different insertion strategies in Figure 4. We insert to the found position if it is a gap (Figure 4a). Otherwise, we insert the key by shifting neighboring keys to the closest gap position to keep the key as close to the prediction position as possible (Figure 4b). This gives us a more efficient exponential search. We account for the accuracy penalty from shifting by incrementing the error in the shift direction. We limit the number of shifts to 30% of the SWseg size (each segment has a unique size) to prevent the accuracy from degrading. Once we exceed this limit, data is inserted into the buffer.

We optimize for the update performance by reducing memory reallocation cost by limiting shift distance to buffer length. Therefore, in the worst case, our insertion cost equals the buffer size B. If the shift distance is longer than the buffer length, we insert the key into the buffer (Figure 4c). Additionally, we support extrapolation insertions, where the insertion in the form of appends may lie outside the segment range (Figure 4d). To avoid over-expanding and generating excessive gaps, we insert the key into the buffer (Figure 4e) if the gaps exceed the maximum gap threshold (50% of the initial segment size). Once the buffer reaches capacity, the segment is flagged for retraining.

4.2.2 Deletion. Invalid data is implicitly erased during search, scan, and insertion. SWIX uses a timestamp threshold to determine if data has expired. The current timestamp is compared to the threshold for data validity. SWIX can also support tuple-based windows (mentioned in Section 2.1) by using the tuple's sequence order as timestamps. For example, the sequence order for n data would be {0,1,...,n-1}, where n is greater than window size W. The current time is n-1 and the latest valid timestamp is equal to $((n-1)-W)$. To explicitly delete, SWIX first follows a lookup procedure, before the found data is converted to a gap in SWarray or erased from the buffer. If a segment expires, we flag its position in SWmeta as a gap.

4.3 Bulk-load

The rationale of the bulk-load algorithm is to segment the data such that the error is minimized when the data distribution is unknown (*cold starts*). Since we use linear models, the segmentation should try to make the data distribution as linear as possible. We propose a second derivative segmentation algorithm using a top-down approach, shown in Algorithm 1. The objective is to find optimal split points such that each segment is linearly predictable while minimizing the number

Algorithm 1 Segmentation Algorithm for Bulk-load

```
1: procedure Split_Second_Derivative(K)
2:     Input: K: collection of keys, where |K| is the number of keys.
3:     Output: Split: collection of index in K to split.
4:     d²(pos)/d(k)² ← second derivative for x = [k_0, k_{|K|-1}], y = [0, |K| − 1]
5:     while N < N_threshold do
6:         pos = max(d²(pos)/d(k)²)
7:         if pos = 0 then
8:             break
9:         end if
10:        Split.push_back(pos)
11:        d²(pos)/d(k)² [pos] = 0
12:        d²(pos)/d(k)² .penalize_neighbors(pos)
13:    end while
14: end procedure
```

of segments to reduce the memory overhead of SWmeta. The insight is that second derivatives capture abrupt distribution changes in a function. Splitting segments at maximums can effectively reduce the errors in each segment.

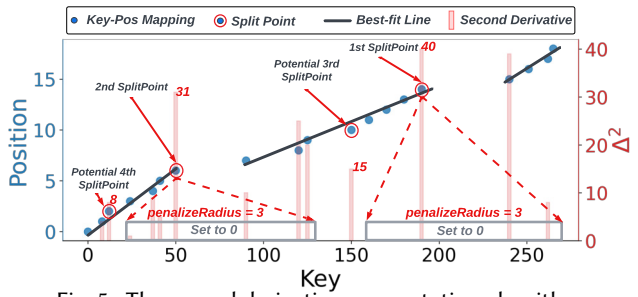


Fig. 5. The second derivative segmentation algorithm.

We illustrate the algorithm with an example in Figure 5. The algorithm first recursively locates positions with the highest second derivatives (shown by the red y-axis). We neglect neighboring points to prevent the segments from being too short. For example, we neglect the point with the second highest second derivative (39) as it is too close to the 1st split point. The algorithm stops at a maximum of $N_{threshold}$ segments. We use 0.01% of the data size n as the default $N_{threshold}$, this was heuristically shown to minimize the number of segments while ensuring the average accuracy. We set an early-stop condition if all remaining second derivatives have similar values to reduce further memory consumption (e.g., if all remaining second derivatives are zero, then one segment is sufficient to model the remaining data with minimal error).

4.4 Retraining

For existing learned indexes, retraining aims to correct the prediction error due to distribution changes. For example, ALEX [9] corrects the prediction model and resizes itself to be 60% full, similar to rebalancing a node in a B+Tree. For the goal of space- and time-efficiency, SWIX minimize the size of the index for the current data distribution along with correcting for any prediction errors. We do not have set bounds on how full or empty a node has to be, but rather extend and shrink

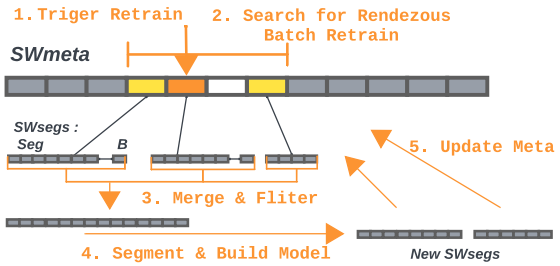


Fig. 6. The rendezvous batch retraining process.

SWarray according to the distribution. For example, we allow more densely packed SWarray when the distribution is uniform and allow for more gaps when the distribution becomes non-linear. SWIX's auto-tuner monitors changes in distribution and adjusts the nodes while retraining.

4.4.1 Retraining SWseg. Retraining the SWseg corrects for prediction errors and merges the buffer with SWarray. To prevent fragmenting the segments during retraining, SWIX introduces rendezvous batch retraining using a bottom-up approach, shown in Figure 6. It is a data distribution aware segmentation algorithm where neighboring segments are retrained together. This approach reduces the average error in each segment compared to splitting a full segment in half or merging two half-empty segments.

When a segment triggers a retrain (step 1), we find all neighboring segments that also need retraining (step 2). We merge and filter all retraining segments (step 3) before locating the optimal split points with a one-pass segmentation algorithm to find positions with abrupt changes (step 4). We split based on a *tol* parameter representing each segment's tolerable error. *tol* controls the size and number of gaps in each segment and is adjusted by the auto-tuner.

To avoid memory overhead from data fragmentation, we do not retrain a segment if no neighboring segments need retraining. This allows them to wait for their neighbors and prevents fragmenting single segments. However, the second time a segment needs retraining, we force retraining to ensure the accuracy of the SWseg.

4.4.2 Retraining SWmeta. Based on a bottom-up approach, modifications on SWsegs during retraining need to be propagated to SWmeta to maintain consistency (step 5 in Figure 6). This involves deleting old segments and inserting new ones. Old segments are marked as expired in the SWmeta, while insertions use the algorithm described in Section 4.2.1. The SWmeta retrain procedure mirrors steps 3-4 in Figure 6, except for the merge process in step 3 (no buffer in SWmeta) and segmenting process in step 4 (no segments needed).

We apply a more relaxed retrain condition to SWmeta because insertions into SWmeta are relatively infrequent and are mostly replacements rather than new insertions. Therefore, SWmeta is allowed to extend and generate more space for insertions more freely by scaling the prediction model. This also limits the shift distance, which reduces the shift cost. We retrain SWmeta when the gaps exceed 20% of the meta size to prevent excessive gaps from extensions and deletions. The retraining algorithm then corrects the prediction error and compacts SWmeta by initializing the number of gaps to be 5% of the SWmeta size after retraining.

4.5 Analytical Evaluation and Auto-tuner

4.5.1 Analytical Cost Model.

$$C^{SWIX} \begin{cases} \log(\epsilon_{meta}) + \log(\epsilon_{seg}) & C_S \\ C_S + \alpha B & C_I \\ \beta_{seg}(S_i + N) + \beta_{meta} N & C_R \end{cases} \tag{1}$$

SWIX is designed as a memory-efficient structure that executes operations efficiently. We provide an analytical evaluation of SWIX's operations, including search (C_S), insert (C_I) and retrain (C_R), from an execution time cost perspective. The evaluation helps us locate optimizations and offers insight into increasing the adaptability of the index when faced with drifts during execution. We now explain the underlying intuition behind each component.

The search cost combines two components: SWmeta search and an associated SWseg search. Given that the search is an error-based operation, the search cost is bounded by ϵ_{meta} (meta error) and ϵ_{seg} (segment error). Generally, these have a negative correlation. To illustrate, to get ϵ_{seg} small the number of segments may need to increase, thereby potentially raising ϵ_{meta}. Both errors are influenced by the *tol* parameter, so choosing *tol* appropriately is crucial to achieve optimal search performance.

Apart from the cost of searching for the insertion position, the main insertion cost is derived from the maximum shift distance encountered when gaps are not available to place the data. SWIX uses a buffer to compensate and facilitate the update costs by limiting the maximum shift cost to the buffer length B in SWseg. Therefore, the insertion cost is constrained by B. Given that the buffer size remains constant to optimize cache efficiency, α is the ratio of the number of keys that fail to be inserted into the predicted location. α is controlled by *tol*, where larger values of *tol* reduce the failure rate and the likelihood of shifts, impacting the insertion performance.

The implicit deletion strategy amortizes the deletion cost across other operations. For explicit deletes, the cost would be the same as the lookup. Any gaps, generated by training or deleting, will be purged from the index during retraining operations. Specifically, the purging cost is included in the scan cost during merging & filtering (step 3 in Figure 6). The retrain cost includes the SWseg and SWmeta retrain, which are bounded by their size, respectively. After SWseg retrains, the newly formed segment(s) are inserted into SWmeta. The insertion cost is the number of segments (N) as the update requires shifting the entire SWmeta from start to end in the worst case. As the frequency of segment retraining (β_{seg}) increases, so do the insertions into SWmeta. As SWmeta is updated more often, the frequency of meta retraining (β_{meta}) also increases, which in return increases the overall retrain cost. β_{seg}, the size of segment i (S_i) and the number of segments (N) are all influenced by *tol*. E.g., a smaller *tol* leads to an increased number of smaller SWsegs, thereby increasing the likelihood of β_{seg}. Conversely, a larger *tol* would have the opposite effect.

The evaluation based on the cost model of SWIX shows that the tuning *tol* is crucial as it affects all operation performance. Optimizing *tol* on the fly is therefore critical to ensuring competitive performance when dealing with dynamic workloads and changing distributions.

4.5.2 Auto-tuner. To capture how different internal states affect the performance of the index in terms of *tol*, we derive a simple heuristic model, where each element in the cost model represents the cost of an operation. Specifically, we carry out multiple experiments with a variety of datasets and index configurations to explore the relationship between *tol* and the actual performance (time cost). We monitor the actual states from different operations, such as search, scan, and shift distances. We use the Akaike Information Criterion (AIC) [2] as a metric to determine the significance of states through step-wise selection. The aim of using AIC is to select a model that best captures the underlying pattern with the least complexity, ensuring robustness in performance predictions. This

involves a process of iteratively adding and removing states for comparison. The primary states we select include average search distances (SWseg & SWmeta), neighbor traversals, and length of merge and retrain. We formulated a heuristic model by fitting these states into a linear model as follows:

$$C = w_{s1}\log(\overline{Search}_{meta} + \overline{Search}_{seg}) + w_{s2}tol \\ +w_{sc}\overline{Ptr} + w_r(Length_{merge} + Length_{retrain}) \tag{2}$$

Where, $w_{s1}, w_{s2}, w_{sc}, w_r$ are the weights for characterizing the significance of the search ($s1$ & $s2$), scan (sc), and retrain (r) operations to the overall cost, respectively. The weights are found by adjusting and comparing the model prediction with the actual performance measurements. For the search cost, the $\overline{Search}_{meta}, \overline{Search}_{seg}$ are the exponential search length in the meta and segment, respectively. Additionally, *tol* also affects the search cost as it is proportional to the initial error of each segment after retraining. We use the average number of sibling traversals $\overline{Ptr}$ to capture the scan cost during range queries. For insertions, the insertion cost itself is related to the shift distance. It is controlled by the buffer length B, which unsurprisingly the AIC process shows no correlation with *tol*. Lastly, $Length_{merge}, Length_{retrain}$ represents the total merge and retrain length, respectively. The total length accounts for the number of retrains (which we seek to minimize).

These states can serve as the bridge between *tol* and performance so that we can adjust the *tol* to influence these states and consequently optimize performance during the run-time. The tuning algorithm starts with a default *tol* and adjusts it based on a tuning rate. The auto-tuner monitors the states and estimates the time cost based on the current state measurements, which saves overheads from adding timers. We tune the *tol* once we reach the tuning rate by adding/subtracting *tol* with the tuning step. The tuning process is inspired by the gradient descent process, where we perturb *tol* in one direction and record its change in cost over updates. If the cost reduces, we carry on in the same direction, otherwise, we move in the opposite direction. We use step decay to reduce oscillations between two *tol* values. If the minimum changes due to distribution shifts, we increase the stride of the tuning step if it continuously moves in one direction.

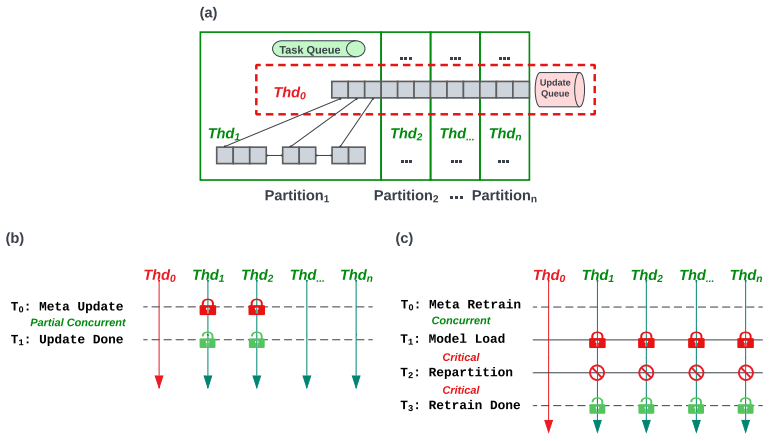


Fig. 7. Parallel SWIX: (a) Overall structure. (b) Synchronization of meta update. (c) Synchronization of meta retrain.

5 PARALLEL SWIX

SWIX supports concurrency with minimal modifications. Parallel SWIX is designed to run in parallel where each thread performs operations simultaneously on a subset of data.

5.1 Parallel SWIX Structure

As Figure 7a shows, the main thread thd_0 manages the whole SWmeta and each of the remaining worker threads handles a number of segments. The number of segments in each partition is static, however, the number of data in each partition varies as the segment size is non-static. We ensure the error is similar between partitions as SWIX's performance is error-dependent rather than size-dependent. Therefore, load-balancing is ensured by having uniform errors across partitions. thd_0 handles all cross-partition modifications and orchestrates the worker threads during updating, retraining, and re-partitioning of SWmeta. SWIX's structure is inherently suited for parallelization. Parallel SWIX partitions the data for different threads. We add concurrency control measures to avoid race conditions while minimizing the synchronization cost. Segments are connected within each partition and disconnected between threads to prevent concurrent data access. Each thread except thd_0 will have a task queue from the dispatcher and works asynchronously to increase throughput. thd_0 uses a queue to receive update requests from the rest of the threads.

5.2 Parallel Search

Searching is done asynchronously. The core search procedure follows that of the *predict-correct* process of SWIX. All threads receive the same search query. Each thread checks if the error bound of the prediction overlaps with its partition. If so, it runs the correction process. If multiple threads overlap with the error bound, we reduce the search space by confining each thread to search within its partition. Therefore the search cost in SWmeta for each thread is at most $O(\log(\min(\epsilon_{meta}, P)))$, and the cost of the consecutive *predict-correct* process in SWseg is $O(\log \epsilon_{seg})$. Where, ϵ_{meta} and ϵ_{seg} are the meta and segment errors respectively. P is the partition size.

For point queries, only one thread will find the result. However, for range queries, several threads may be involved as the result may span across multiple partitions. In this case, one of the threads will find the lower-bound of the range, and other threads that contain records between the lower bound and upper bound of the range perform a scan through their partition. We stop when reaching the upper-bound. The final results are aggregated asynchronously [2].

5.3 Parallel Updates

Inserting keys into segments is asynchronous and involves a parallel look-up followed by a serial insertion at the segment level. The insertion cost in each thread is $O(\log(\min(\epsilon_{meta}, P)) + \log \epsilon_{seg} + \alpha B)$, where B is the buffer length. Deletion is done implicitly during other operations.

5.4 Concurrent Retraining

Retraining Parallel SWIX is similar to SWIX, but Parallel SWIX assigns the retraining of each segment to its corresponding thread.

5.4.1 Retraining SWseg. Once a segment triggers a retrain, the thread tries to find neighboring segments to retrain together, similar to SWIX's rendezvous retrain strategy (Section 4.4). We only allow batch retraining within the same partition to avoid additional synchronization. Each thread retrains its segments asynchronously.

[2]Depending on the query, we aggregate the results using a thread-safe (e.g., `C++ atomic`) variable for count queries and use a pre-allocated array of pointers if keys are to be returned. Specifically, the size of the array is the $number\ of\ threads$ - 1, where each thread asynchronously returns the result.

Index Type	Index	Metrics	SYN								REAL							
			lognormal		normal		udense		usparse		books		fb		osm		wiki	
			AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
No Index	Queue	*Overhead (%)*	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		Latency (µs)	2857.8	5664.1	2794.4	5537.4	2809.8	5568.9	2793.1	5534.1	2790.6	5532.4	2817.2	5583.9	2807.4	5562.4	2840.7	5630.4
Alogrithmic Index	B+Tree	*Overhead (%)*	72.2	130.7	67.1	130.5	66.9	130.5	72.3	130.7	72.2	130.7	66.9	130.5	67.1	130.5	66.9	130.5
		Latency (µs)	27.1	268.2	26.3	259.1	27.7	274.1	27.0	266.2	27.1	265.9	26.5	261.4	26.2	255.3	26.9	265.94
Streaming Index	IMTree	*Overhead (%)*	35.2	54.8	35.2	54.8	35.2	54.8	35.2	54.8	35.2	54.8	35.2	54.8	**35.2**	**54.8**	35.2	54.8
		Latency (µs)	25.3	246.9	24.9	242.5	24.7	240.1	24.6	239.9	24.6	240.8	**24.4**	**238.6**	**25.7**	252.7	24.5	238.0
Updatable Learned Index	PGM	*Overhead (%)*	124.9	143.8	124.8	143.8	124.4	142.0	124.5	142.0	125.0	144.6	125.4	144.0	125.3	143.4	125.0	144.2
		Latency (µs)	32.6	315.6	33.1	321.0	32.3	312.8	32.4	312.7	32.6	316.0	33.3	323.3	33.5	323.7	32.7	315.6
	ALEX	*Overhead (%)*	45.6	56.7	43.4	55.3	55.9	64.1	55.9	64.1	56.0	64.1	50.0	53.1	102.8	111.3	40.5	47.3
		Latency (µs)	19.4	188.8	**18.6**	**180.8**	28.0	282.0	20.1	195.7	24.7	245.1	32.2	317.3	31.4	310.2	24.7	247.1
Streaming Learned Index	SWIX	*Overhead (%)*	**7.0**	**6.3**	**7.2**	**6.5**	**6.8**	**6.1**	**7.0**	**6.2**	**6.8**	**6.1**	**32.4**	**48.9**	39.1	56.3	**14.3**	**19.0**
		Latency (µs)	**19.2**	**186.5**	19.6	190.9	**18.8**	**182.4**	**18.5**	**179.0**	**18.3**	**176.8**	25.0	245.0	26.0	**252.6**	**22.8**	**224.2**

Fig. 8. Memory overhead $\frac{total\ size - data\ size}{data\ size}$ and latency $\frac{total\ time}{total\ updates}$ for each experiment setting excluding the ordered workload. Note: MAX for the memory overhead is measured when the total memory consumption ($total\ size$) is at its largest and can be smaller than the AVG.

Following the SWseg retrain, SWmeta needs to insert the newly formed SWseg(s) that have been split and delete old SWseg(s). The local thread flags the expired SWsegs for deletion. Insertion is tricky because the nearest gap might be in another partition. Therefore, the thread first looks for gaps within its own partition. If it finds any, it inserts into SWmeta without synchronization. Otherwise, it flags the insertion position and delegates the insertion task to thread thd_0 through the meta update queue and moves on to the next task. Synchronization is unavoidable if the thread needs to access the segment before thread thd_0 completes the insertion. In this case, it waits until thd_0 finishes the insertion before proceeding.

Once the meta thread finds the nearest gap, it synchronizes all threads that overlap the shift distance before shifting the segments toward the gap position. The synchronization is shown in Figure 7b, where the insertion position is located in thd_1. The meta thread locates the gap and blocks thd_2 before blocking thd_1 for shifting.

Synchronization is also needed when inserting segments at the partition boundaries. For example, a segment can be inserted either at the end of thd_1 or the beginning of thd_2. In this case, thread thd_0 handles the insertion and chooses the position that minimizes shifting. Then, it blocks the thread that owns that position.

5.4.2 Retraining SWmeta. Retraining SWmeta requires stopping and synchronizing all threads, as illustrated in Figure 7c. To reduce the critical section and prevent long stalls, thd_0 retrains the model on a snapshot and lets other threads run without blocking. After the model is trained, thd_0 blocks all threads, creates a new SWmeta, and redistributes the segments.

6 EXPERIMENTAL EVALUATION

The experiments evaluate SWIX's auto-tuner, and error distribution, before comparing SWIX's performance against the competitors, including serial and concurrent indexes under various streaming parameters and workloads. The highlights are as follows:

- We verify the effectiveness of the auto-tuner and show it can optimize *tol* for different distributions.
- For a default IBWP workload, the memory size of SWIX is, on average, 22.22% of a B+Tree, 42.79% of an IMTree, 12.06% of a PGM-Index, and 25.91% of ALEX.

- At the same time, SWIX attains a speed up of 1.30× speedup over B+Tree, 1.20× over IMTree, 1.59× over PGM-Index, 1.19× over ALEX and 136.55× over Queue.
- For a concurrent workload, Parallel SWIX uses 41% and 34% of the memory consumption of FINEdex and XIndex, while achieving up to 1.90× and 3.45× more throughput for update-heavy workloads, respectively.

6.1 Experimental Setup

SWIX is implemented in C++. Experiments are conducted using the gcc compiler on a 16-core machine with Intel E5-2690@2.60GHz and 64GB of RAM running Ubuntu 14.04.

6.1.1 Dataset. We use four synthetic datasets (*lognormal*, *normal*, *uniform dense* (*udense*) and *uniform sparse* (*usparse*)) and four real-world datasets (*books*, *fb*, *wiki* and *osm*) from the SOSD benchmark [21]. The datasets contain 1-dimensional values used as keys in the primary index. We generate the timestamps[3] to evaluate arrival/departure from two distributions for different workloads (Details in Section 6.1.3). We can evaluate the index performance under different data distributions with these datasets. Synthetic datasets tend to have locally linear distributions, while real-world datasets show much more non-linear behavior. We show the two representative datasets (*udense*, and *osm*) in the comparative analysis due to the space limit. *udense* has a uniform local and global distribution, while *osm* is non-uniform in both local and global data distribution. Results containing all datasets are summarized in Figure 8 and the detailed plots can be found here[4].

6.1.2 Baseline. The baseline includes an array-based queue, STX B+Tree [4], STX B+Tree implementation of IMTree [35], the Dynamic PGM-Index [11, 14], ALEX [9, 31], and FLIRT[4] [44]. To enable a fair comparison, we parameter sweep the fanout (B+Tree) and error parameter (PGM-Index) in a range from 4 to 4096 and report the lowest latency results for each experiment. The size of the IMTree insertion tree is $0.15n$, following [35]. For concurrent learned indexes, we use XIndex [39, 40], FINEdex [26, 38], and PPFlirt as the baselines. PIMTree [35] is not included as no open-source implementation is available.

We add a secondary index to B+Tree, PGM-Index, ALEX, XIndex, and FINEdex for handling IBWP workloads (see Section 2.2). The secondary index includes a mapping from timestamp to key in a sorted queue. We did not include the memory and search cost of the secondary index in the experiment to be fair to the baselines.

6.1.3 Workload. We simulate an IBWP workload with a sliding time window. The window length W is the time frame for which data exist in the window, and the data size n is the number of data in the window. Data does not have to enter the window at every timestamp, and multiple data can arrive at the same time. Therefore, n correlates to W but may fluctuate over time.

For the default workload, the window length is $10M$, and we perform $40M$ updates to the window length[5]. The keys are shuffled to ensure random updates. Each update moves the window by one step and triggers a search query. The default workload uses a range query with a scan range of 1000 keys. To unify the data size, we assume that the initial window before updates has one data at every timestamp such that $n = W = 10M$. During the updates, the timestamps are uniformly distributed such that the average data size is equal to the time frame.

We test the arrival/departure rates for the skewed workload by generating timestamps from a normal distribution $N(\mu, \sigma^2)$ with varying values of σ^2. Smaller variances result in data burst,

[3]Streaming benchmarks YSB [6] and Stock [46] showcase how streaming systems schedule different arrival and departure rates. We omit these benchmarks because the timestamps are uniformly distributed.
[4]https://github.com/SWIXProject/SWIX
[5]M denotes *millions*, K denotes *thousands*.

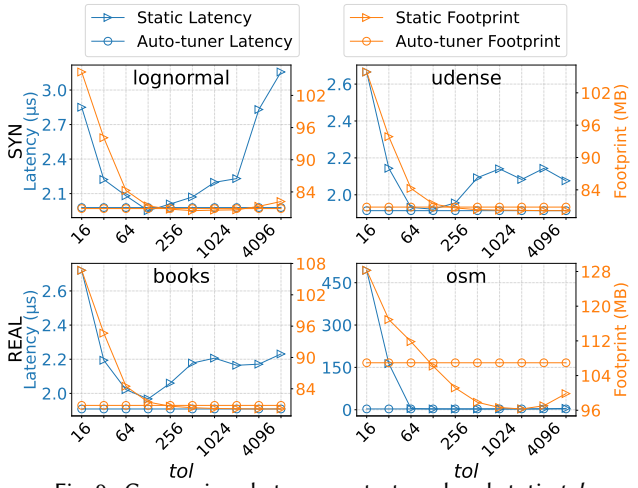


Fig. 9. Comparison between auto-tuned and static *tol*.

while larger variances spread the timestamps more evenly. Therefore, in the skewed workload, n varies with σ^2, while W remains constant.

6.1.4 Performance Metrics. We measure *memory usage* in *MB* which include the index and data size. Time performance is measured as *latency* in μs and *throughput* in ops/s using a *Read Time-Stamp Counter* (RDTSC). RDTSC instructions minimize the overhead for time measurements, accounting for less than 5% overhead [20, 46]. Latency and throughput are reported for one window slide, which includes both time for search and update.

6.2 Detailed Analysis

6.2.1 Evaluation of Auto-tuner. To verify the effectiveness of the auto-tuner, we run SWIX with static values of *tol* (ranging from 16 to 4096), and compare it with auto-tuned *tol*s. For the auto-tuner, we tune the index once whenever 10% of the window has slidden. The results are shown in Figure 9. Synthetic datasets are displayed in the top row, and real datasets are at the bottom. The latency is in blue (left y-axis), and memory usage is in orange (right y-axis). The auto-tuner results are shown as horizontal solid lines. Memory usage and latency look convex when varying *tol*, which suggests there is a minimum. The near-optimal latency performance validates the ability of the heuristic model to capture time cost and supports the use of the auto-tuner. Since we use a lightweight auto-tuner to reduce computational costs (latency), the auto-tuner could yield a high index size for datasets with strong nonlinearities (*osm*), as our heuristic does not put any constraint on memory footprint. For *osm*, in particular, it tends to trade off space for time by generating a large number of smaller segments to reduce the accuracy penalties. Nonetheless, one can reduce the memory overhead using constrained optimization and add a memory constraint to limit the memory footprint. This approach can reduce the index size for highly non-linear data but increases the latency.

6.2.2 Evaluation of Segment Error. Figure 10 depicts the average segment error ($\bar{\epsilon_{seg}}$) with updates. For *udense*, errors are balanced. However, for local non-linear CDFs (*osm*) segments are extended to generate gaps for insertions, which lead to more rightward shifts as gaps are located towards the right end during extensions. $\bar{\epsilon_{seg}}$'s damped cosine wave pattern demonstrates the auto-tuner's ability to optimize model accuracy. It also explains the memory consumption of *osm*, where the auto-tuner generates more segments to reduce prediction error.

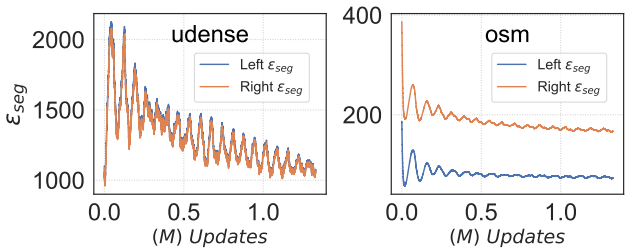


Fig. 10. ϵ_{seg} evolution over updates.

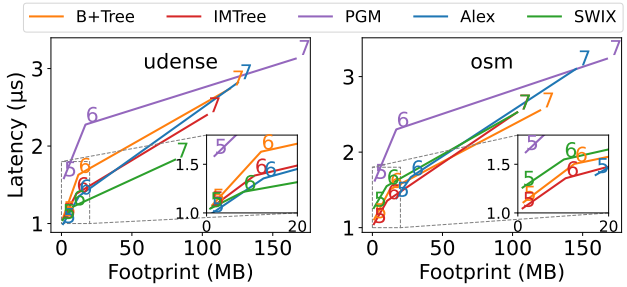
6.3 Comparative Analysis

We compare SWIX with baselines across diverse stream characteristics and workloads, summarized in Figure 8. The fastest index per dataset is colored in dark green, and those with the lowest memory overhead are in light green. SWIX generally has the lowest memory overhead, except for *osm* (where all learned indexes underperform). Notably, SWIX is memory-efficient with large datasets, as its overhead at MAX often matches or even beats the AVG overhead.

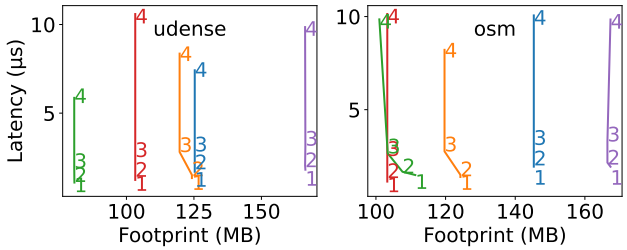
Generally, tree structures are memory-intensive, as evidenced by the B+Tree. Interestingly, although IMTree employs two trees, it reduces memory usage by decreasing the tree height of the B+Tree. The static search tree achieves near-complete compaction (99% occupancy), while the smaller insertion tree has a 71% occupancy, it still outperforms a single B+Tree with 63% occupancy. Learned models can condense data which significantly reduces the tree structures' memory overhead, as exemplified by ALEX. However, using numerous sub-indexes to further reduce the height of trees, as seen in the PGM-Index's log-merge tree design, increases memory overhead. SWIX, on the other hand, truncates tree height to two without having multiple sub-indexes and leverages learned models to compact the top layer, thereby considerably cutting down memory usage. However, the ability of the learned model to compact the data hinges on the data distribution, as exhibited by the fluctuating overhead for different datasets.

Similar to the memory findings, learned indexes do not perform particularly well with real-world datasets, especially *osm* and *fb* when the distribution is hard to learn, making them highly data-dependent. In contrast, B+Tree and IMTree are data distribution agnostic (as expected). Interestingly, out of the three learned indexes, PGM-Index is shown to be distribution independent, which suggests that log-merge tree structure can effectively buffer distribution changes. ALEX uses a cost model to determine when to split a node, but its main inefficiency is with the split mechanisms. ALEX expands nodes to be 60% full (40% are gaps), to reduce retraining costs. This is memory inefficient for internal nodes, as it expands the number of pointers without changing the number of child nodes. Nodes are also allowed to split downwards, which generates more internal nodes and increases the height of the tree. In contrast, SWIX only has one internal node (SWmeta) and only allows at most 20% of SWmeta to contain gaps (compacted to 5% when retrained). Furthermore, ALEX splits a full data node in half, which disregards the distribution in the segments and may result in data fragmentation (space and time inefficient). SWIX opts to retrain the segments together to avoid data fragmentation and merge segments to ensure the size of the SWmeta node. Therefore, SWIX has the lowest memory overhead and very competitive performance in real and synthetic data. The results also suggest that the two-layer structure does not compromise the speed up from learned models.

In the following sections, we analyze the performance under stream characteristics (window lengths, scan range, and arrival rates) and workloads on the two representative datasets.



(a) Time (μs) and space (MB) trade-off of different window sizes.



(b) Time (μs) and space (MB) trade-off of different scan ranges.

Fig. 11. Analysis of different window sizes and scan ranges.

6.3.1 Window Length. We vary the window length from $100K$ to $100M$ in Figure 11a to represent varying data sizes. The numbers in the figures represent the exponents of window lengths ($10^5, 10^6, 10^7$). The main highlight is that SWIX achieves the highest performance-to-memory, evidenced by being closest to the bottom left corner of the figure. For larger W, the advantage of SWIX is more noticeable compared with other learned indexes.

SWIX performs very well in both space and time for easier-to-learn distributions (*udense*) (shown in Figure 9). For *udense*, the average segment exponential search length increases by 1.36× with increasing window length. The number of segments increases by roughly 4.85×; this is marginal compared to the data size increases of 100×. The increase in latency over data sizes is due to the retrain length increasing by 34×, as more data needs to be merged and retrained. The auto-tuner aims to minimize the latency and does not cap the memory constraint. Therefore, it generates more segments which drastically increases space over data size: a 63× increase in segments from a W of 10^5 to 10^7 for *osm*. Furthermore, the average segment exponential search length increases by 1.66×, a 22% increase compared to *udense*, as variation due to non-linear data distribution is more prominent at larger W.

The baselines do not scale well with increasing data size. The B+Tree, IMTree, and PGM-Index are not dependent on data distribution, while ALEX suffers from hard-to-learn data distributions. For *osm*, ALEX generates 657 model/internal nodes when the window length is $100M$, which creates a very deep structure. In contrast, the B+Tree only has 232 internal nodes. ALEX not only suffers for *osm*, it unexpectedly underperforms when the CDF is entirely uniform (*udense*). For *udense*, ALEX is structurally two-layered. This allows for a direct comparison between SWIX and ALEX, where SWIX is shown to be smaller and faster.

In summary, SWIX's space and performance are shown to be an error-bounded problem rather than a space-bounded one, which means it can scale exponentially with easily-to-learn distributions.

6.3.2 Scan Range. We simulate short and long-range queries by varying the scan range from 10^1 to 10^4 (Figure 11b). For most indexes, only time efficiency depends on the scan range, evidenced by the vertical lines. This experiment shows the limitations of implicit deletes: memory overhead increases with shorter-range scans as fewer keys are traversed in the segment. Again, this is more prominent for hard-to-learn datasets (*osm*). Regardless, SWIX is still the most space efficient. As for baselines, IMTree and B+Tree are independent of data distribution. B+Tree differs in size for each scan range because we use parameter sweep to optimize the fanout for latency. PGM-Index and ALEX are shown to be memory inefficient despite the fact that they are learned indexes, which shows that their design is focused on speed rather than size.

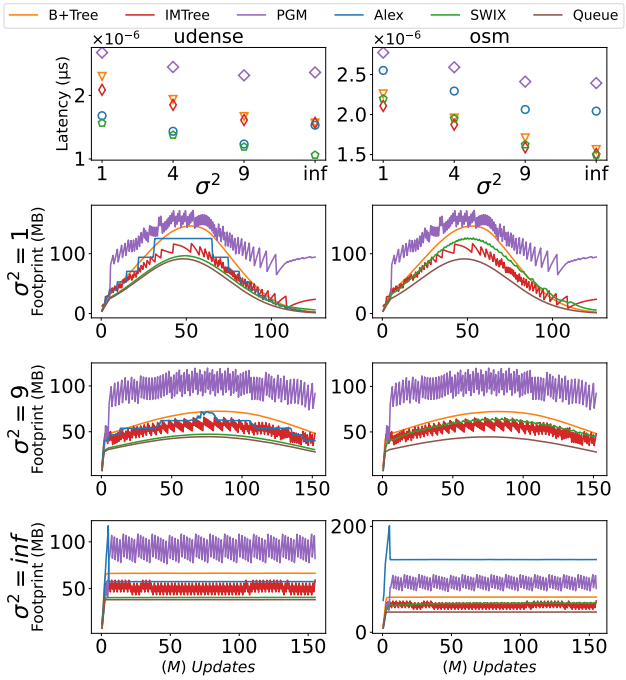
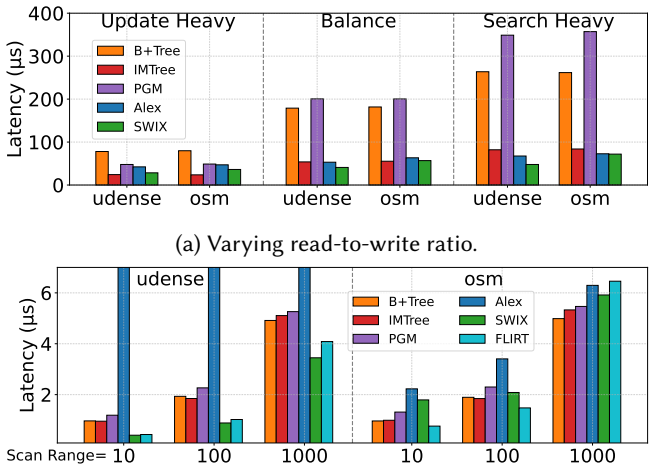


Fig. 12. Latency (μs) and Index footprint (MB) for different degrees of σ^2 in timestamps.

6.3.3 Skewness in Timestamp. Streaming indexes must handle efficient data bursts, which cause the data size to change rapidly. During data bursts, the data size quickly expands as the insertion rate surpasses the expiration rate. The data size then reduces after the burst with more deletions. We simulate different arrival/departure rates with the procedure described in Section 6.1.3. σ^2 is varied from 1 to inf, where smaller σ^2 represents a sharp data burst and σ^2 = inf means uniform timestamps. We present the results in Figure 12. The top row shows time efficiency against σ^2. The bottom three rows show a temporal view of the memory footprint over updates for $\sigma^2 = 1, 9, \text{inf}$. Note that ALEX fails to run for memory test for *osm* when $\sigma^2 = 4, 9$. We have shown in Section 6.3.1 that the auto-tuner can scale well with data sizes. This evaluation highlights the auto-tuner's ability to scale with changing index size during runtime, evidenced by SWIX's low memory overhead and latency.

The experiment highlights the smooth memory profile of SWIX during size transitions, even for extremely sharp data bursts, which also explains why MAX overhead is close to the AVG overhead

in Figure 8. This is attributed to the flexible structure that adjusts the size of SWarray and the number of gaps according to data distribution, which greatly reduces wasted space. This design is very different from the baselines, which use predefined structures. B+Tree rebalances the node when it is 50% empty and 100% full. PGM-Index and IMTree have saw-teeth spikey patterns because they batch the expiry operations. Therefore, if the merge criteria are not met, expired data will not be erased, making the index less compact. This effect can be seen from the tail pattern from the low-variance figures (second row). ALEX bounds the number of gaps between 20% and 40%. The lagging/step-wise effect is seen most prominently in ALEX, where it tends to reserve more space when data size increases and does not immediately compact the index. It also tends to reserve too much space as shown in the spike in $\sigma^2 = \text{inf}$ case. In terms of speedup, ALEX can achieve higher speedup compared to SWIX at the expense of memory, however, both suffer from hard-to-learn datasets (*osm*).



(a) Varying read-to-write ratio.

(b) Ordered data.

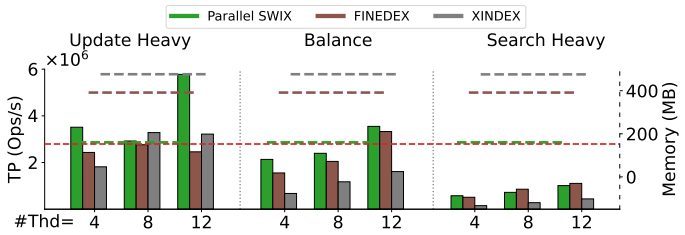
Fig. 13. Comparison of learned indexes on different workloads.

6.3.4 R-W Workload. We investigate the time performance of SWIX under different read-to-write ratio (RW ratio) in Figure 13a. We omit the memory overhead for the RW ratio workload since they are identical to the previous analysis. We show three workloads (from left to right): an update-heavy workload with a 1:10 RW ratio, a balanced workload with a 1:1 RW ratio, and a search-heavy workload with a 10:1 RW ratio. Since each one has different numbers of reads and writes, we keep the updates constant at $400M$ as the anchor. Therefore, the read-heavy workload will have 10× more search operations than the balanced case. SWIX achieves the average best time efficiency performance across workloads, especially in the search-heavy case. The performance is dominated by the search time as the y-axis is higher for the read-heavy workload. The search performance benefits from the learned model, while the update performance benefits from the dynamic SWarrays. Although SWIX is updatable for a learned index, it cannot outperform IMTree for the update-heavy workload, as the tree design is very efficient for insertion with no retraining. The trade-off is a slower search time since IMTree must search two trees with no models.

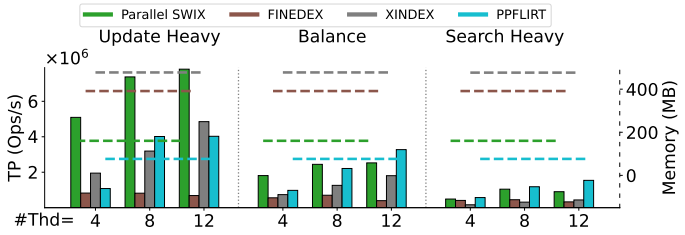
6.3.5 Ordered Workload. We show the performance of the ordered workload in Figure 13b. The datasets are sorted and updates are in the form of appends, mimicking that of FLIRT. PGM-Index

performs much better for this workload and shows competitive performance across datasets. Since data is appended in order, the keys expire from the largest sub-index and are inserted into the smallest sub-index, which allows PGM-Index to function like a queue and efficiently delete entire sub-indexes during merges. ALEX's strategy of expanding the root node before recursively expanding the last node is not shown to be efficient. SWIX performs much better in this situation by expanding the relevant segment without accuracy penalties. However, SWIX still suffers for *osm*, as the auto-tuner tries to balance the performance for the entire index and cannot focus only on the distribution changes occurring in the last segment. FLIRT performs well across datasets, especially when the match rate is small. FLIRT's design ensures the error in each segment, thereby bounding the last-mile search. This results in superior lookup performance even for *osm*, evidenced by the efficiency at small scan ranges. However, this also means FLIRT generates more small-sized segments, reducing the scan performance as more pointers are traversed.

The memory overhead for *osm* are 62.2% for SWIX, 87.3% for ALEX, 119.3% for PGM-Index, 100.4% for B+Tree, 46.2% for IMtree and 22.9% for FLIRT[6]. For non-ordered data optimized indexes, the memory usage increased across the board compared to Figure 8 except for the PGM-Index. This supports the observation above, where the PGM-Index can efficiently function like a queue. FLIRT, which functions as a learned queue, unsurprisingly achieves the lowest memory overhead as it is optimized for this workload, uses a two-layer design, and does not have any gaps or buffers. SWIX and other indexes with mechanism to support random insertions clearly is not suited for this workload. Specifically, SWIX's implicit deletion strategy suffers as not all searches will be at the beginning of the index, causing lingering expired data to persist.



(a) Varying read-to-write ratio.



(b) Ordered data.

Fig. 14. Comparison of parallel learned indexes on different workloads. Bars represent throughput (TP, left y-axis), while the dashed line indicates memory usage in MB (right y-axis).

[6]FLIRT's overhead is calculated over a 1D dataset as FLIRT does not store the key and timestamp separately.

6.4 Parallel Learned Indexes

We evaluate the scalability (in terms of throughput) and memory usage in Figure 14a for varying RW ratios. The memory usage of the three parallel indexes is shown in dashed lines and corresponds to the right y-axis. We show the data size with a red dashed line for comparison.

A key difference between Parallel SWIX and the baselines is the concurrent design. XIndex and FINEdex opt for a concurrent design where threads freely access the index, while we opt for a parallel design where threads handle a subset of the index. The advantage of a parallel design is that each thread has to process less data. Therefore, Parallel SWIX performs much better than the concurrent indexes for low thread counts. To fully take advantage of parallelism, the workload should ideally be isolated to each partition as with updates, where we can achieve much higher throughput compared to concurrent designs in the update-heavy figure. However, if the workload covers multiple partitions, such that multiple threads have to do the same work (e.g., range queries), the efficiency of a parallel design decreases. This can be seen in the search-heavy figure, where FINEdex scales much better compared to SWIX.

Comparing the design between the indexes, Parallel SWIX is shown to be extremely memory efficient compared to the other two baselines, with memory usage only slightly higher than that of data size (similar to SWIX in Figures 12). While all three indexes have two layers, FINEdex uses a B-Tree to store the prediction models in the root layer. At the leaf level, it keeps modified B-trees (with an 82% occupancy) to keep data stored without buffers. Keeping the leaf level sorted improves search performance for range queries shown by the search-heavy and balanced workloads. Both SWIX and XIndex keep the data unsorted and have to search the data array and the buffer for the results, which reduces search time, especially with longer scan ranges. However, using a buffer is more update-friendly.

Lastly, we evaluate the performance of Parallel SWIX in an append workload in Figure 14b. XIndex and FINdex suffer from this workload as their design requires a lot of synchronization as threads are waiting in line to access the first and last segments for updates. Again, PPFlirt achieves the lowest memory usage, as it is optimized for this workload and behaves like a queue. It is shown to perform especially well with search-heavy workloads, as it keeps the data sorted and does not have gaps. Similar to Parallel SWIX, PPFlirt uses a parallel design which takes full advantage of this workload where updates are isolated in the front and back partitions. However, PPFlirt suffers from update-heavy workloads as the dequeue and enqueue threads move between threads which requires synchronization. Parallel SWIX does not require any synchronization during updates unless there are changes to the SWmeta.

In summary, for a general workload, Parallel SWIX achieves the lowest memory usage and competitive performance against FINEdex and XIndex. The parallel design is able to reduce the work in each thread. However, it struggles when multiple threads must execute the same task. For an append workload, Parallel SWIX is able to achieve competitive performance for update-heavy workloads against PPFLirt, with the cost of extra memory usage.

7 RELATED WORK

7.1 Traditional In-memory Indexes

Classical in-memory indexes include hash tables, B+Trees [13, 25], skiplists [36, 42], and tries [5, 24]. Most databases use B+Tree and its variants [13, 24, 25] as the de-facto standard, which can handle both point and range queries [10]. Tree-based index structures such as B+trees take considerable memory as they tend to be generic and do not consider the data distribution. Therefore, their performance can degrade substantially on large datasets due to pointer chasing, which incurs

multiple cache misses [13]. Additionally, traditional indexes are not efficient for enqueue-dequeue operations in IBWP workloads, as they are not designed for such patterns of updates.

7.2 Learned Indexes

Learned indexes [23] build models to capture the Empirical Cumulative Distribution Function (ECDF) of the indexed key. To look up a key, the index predicts the physical location of the records. Much work has been done on learned indexes for both single-dimensional data [9, 11, 12, 17, 18, 22] and multi-dimensional data [15, 19, 27, 32, 33].

Learned indexes can use different predictive models, including linear and non-linear models [30]. Experiments have shown, however, that simple linear models are effective for most real-world datasets [21, 29]. As a result, most learned indexes use linear spline models and organize records into linearly-predictable segments.

For workloads involving updates, it is crucial that the learned index can efficiently be updated after the model has been trained. A number of learned indexes have been suggested to handle updates, including PGM [11], ALEX [9], and others [7, 16, 26, 28, 34, 40, 41, 45, 47]. However, no existing learned index can process IBWP workloads without external structures or additional lookups, which substantially increases the memory footprint.

Moreover, the choice of the higher-level structure plays a key role in the performance of the learned index structure. Read-only indexes such as RMI [23, 30] and RadixSpline [22] use array-based contiguous memory blocks to access data at the cost of being read-only. To tackle this issue, updateable learned indexes such as ALEX [9] use hierarchical tree-based structures to allow manipulation of part of the index without having to retrain the entire model. In terms of segmentation, existing works either are based on a top-down approach (e.g., ALEX [9] and RMI [23]) or a bottom-up approach (e.g., PGM [11] and FITing-Tree [12]). SWIX adopts the top-down approach for bulk-load and the bottom-up approach for retraining.

7.3 Stream Indexes

Some index structures are specifically designed for particular operations on data streams, such as streaming joins [35, 43]. For example, IMTree [35] is an index based on B+Trees used for streaming joins. IMTree keeps track of the keys and timestamps and deletes expired keys periodically when merging B+Trees. As mentioned, the only learned index suggested for streaming data is FLIRT [44]. To the best of our knowledge, there is no streaming index that is specifically designed for IBWP or can be used efficiently for IBWP workloads.

8 CONCLUSIONS

In this paper, we present SWIX, an efficient index for data stream processing based on learned indexes. SWIX is a dynamic, lightweight, parallelizable learned index built upon a flat structure, offering an efficient solution for data stream management. As the experimental evaluation shows, SWIX and Parallel SWIX achieve considerably improved memory saving while maintaining competitive performance across different datasets and workloads compared to existing streaming indexes and updatable learned indexes. Although SWIX is specifically designed to be efficient for IBWP workloads, it can also be used as a general-purpose, memory-efficient index for update-heavy workloads, as we demonstrate this by evaluating SWIX with various read-to-write ratios. This paper not only showcases the prowess of SWIX but also suggests a shift in design paradigms, where we must consider the importance of space efficiency as current design considerations value speed over time. SWIX takes a different approach to consider space as a priority to show the potential of a flat structure in learned indexes (not a trivial task, especially with updates).

9 THEORETICAL ANALYSIS OF SEGMENT COUNT IN FLAT LEARNED INDEXES

To better understand SWIX's performance, we want a theoretical estimation of the prediction error of piecewise linear models. Theorem 9.3 provides this estimation, stating that the (expected) error scales as $O(n^{0.75}/N^2)$, where n is the number of keys, N the number of segments, and the constant factor C depends on the linearity of the data. If the gaps between the keys tend to be similar, C (and the prediction error) should be small.

This scaling law gives theoretical insights into the performance of a flat index structure such as SWIX. As in Section 4.5, denote by ϵ_{meta} and ϵ_{seg} the prediction errors at the meta and segment level, respectively. By Theorem 9.3, we can estimate $\epsilon_{meta} = O(N^{0.75}/1^2)$ and $\epsilon_{meta} = O(n^{0.75}/N^2)$. Hence, the search time is dominated by $\log_2 \epsilon_{meta} + \log_2 \epsilon_{seg} = \log_2(\epsilon_{meta}\epsilon_{seg}) = O(\log_2(n^{0.75}/N^{1.25}))$.

If the constant factor C is small, this represents a significant speed up when compared to binary search and even indexes like B+Tree, whose search time is $O(\log_2 n)$. This is reflected in the experiments with synthetic datasets, where SWIX achieves very low search latency. In the context of well-behaved data, it is not necessary to go beyond SWIX's two layer structure.

For complicated datasets like *osm*, a flat structure still makes theoretical sense. Suppose we add a new layer to SWIX, between the top and bottom layers, alongside a piecewise linear model with $\tilde{N} < N$ segments. By the scaling law for the error, search time would now be dominated by $O(\log_2(n^{0.75}/N^{1.25}\tilde{N}^{1.25}))$. This is better asymptotically than the case with two layers. However, each new layer introduces a constant factor C, which can be large for the case of real datasets. Hence, more layers do not necessarily improve search latency. Given this, we prioritize for SWIX the advantages of a two layer structure, such as a more compact index.

We now provide a proof of Theorem 9.3. We take the assumption that the keys $X_1, \ldots, X_n$ constitute an independent random sample from an underlying distribution with finite variance σ^2. We denote by $X_{(i)}$ the i-th lowest value. Using a well-known result, we can bound the expected sample range.

Lemma 9.1. *Let $R_n = X_{(n)} - X_{(1)}$. Then $\mathbb{E}\left[\sqrt{R_n}\right] \leq \sqrt{\sigma}\sqrt[4]{2n}$.*

Proof. Since the square root is a concave function, by Jensen's inequality we know that $\mathbb{E}\left[\sqrt{R_n}\right] \leq \sqrt{\mathbb{E}\left[R_n\right]}$. It is also known that $\mathbb{E}\left[R_n\right] \leq \sigma\sqrt{2n}$, see for example the discussion after Theorem 5.5.2 in [8]. The result follows from these two facts. □

We use the scaled values $Z_i = (X_{(i)} - X_{(1)})/R_n$. Also, for any $i = 1, \ldots, n-1$, we define the gaps $\Delta Z_i = Z_{i+1} - Z_i$ and $\Delta X_i = X_{(i+1)} - X_{(i)}$. We also assume the $\{X_i\}$ are integers and come from a discrete distribution, which is reasonable in practice. Hence, we have $\Delta X_i \geq 1$, which means $\Delta Z_i \geq 1/R_n$ and we get the following.

Fact 9.2. *It holds that $\sum_{i=1}^{n-1} \frac{1}{\Delta Z_i} \leq (n-1)R_n \leq nR_n$.*

To analyze the error, we do not focus on the $\{X_i\}$. Instead, we consider the equivalent problem of learning the empirical distribution of the $\{Z_i\}$. Specifically, consider any twice-differentiable function $F : [0,1] \to [0,1]$ that interpolates the points $(Z_i, i/n)$. Notice that F captures the empirical cumulative distribution of the $\{Z_i\}$, since $F(Z_i) = i/n$ for all i. Now, denote by $\tilde{F}$ an approximation of F, such that $\tilde{F}$ is a piecewise function consisting of N segments, each a polynomial of degree 1. The predicted index for a key X_i would be $n\tilde{F}(Z_i)$. The average prediction error is then

$$\varepsilon = \frac{1}{n}\sum_{i=1}^{n-1}\left|i - n\tilde{F}(Z_i)\right| = \sum_{i=1}^{n-1}\left|\frac{i}{n} - \tilde{F}(Z_i)\right| = \sum_{i=1}^{n-1}\left|F(Z_i) - \tilde{F}(Z_i)\right|,$$

where we are summing to $n-1$ instead of n for technical reasons (the difference is negligible). Now, the following holds.

Theorem 9.3. *Under the conditions stated in this section, it holds*

$$\mathbb{E}[\varepsilon] \lesssim \frac{\sqrt[4]{2}}{N^2}\sqrt{\frac{\sigma}{120}}\left(\int_0^1 |F''(z)|^{2/5}dz\right)^{5/2} n^{3/4}.$$

Proof. Squaring both sides in the definition of ε we get

$$\varepsilon^2 = \left(\sum_{i=1}^{n-1}\left|F(Z_i) - \tilde{F}(Z_i)\right|\right)^2$$
$$= \left(\sum_{i=1}^{n-1}\left|F(Z_i) - \tilde{F}(Z_i)\right|\sqrt{\frac{\Delta Z_i}{\Delta Z_i}}\right)^2$$

Applying Cauchy–Schwarz inequality

$$\leq \sum_{i=1}^{n-1}\left|F(Z_i) - \tilde{F}(Z_i)\right|^2 \Delta Z_i \sum_{i=1}^{n-1}\frac{1}{\Delta Z_i}.$$

Approximating the Riemann sum by its integral, we get the approximate inequality

$$\varepsilon^2 \lesssim \int_0^1 \left|F(z) - \tilde{F}(z)\right|^2 dz \sum_{i=1}^{n-1}\frac{1}{\Delta Z_i}.$$

Taking square roots at both sides of this inequality and using the result from [3] to bound the integral, we get

$$\varepsilon \lesssim \frac{1}{N^2}\frac{1}{\sqrt{120}}\left(\int_0^1 |F''(z)|^{2/5}dz\right)^{5/2}\sqrt{\sum_{i=1}^{n-1}\frac{1}{\Delta Z_i}}.$$

We bound $\sqrt{\sum_{i=1}^{n-1}\frac{1}{\Delta Z_i}}$ using Fact 9.2, apply expected value at both sides, and bound $\mathbb{E}\left[\sqrt{R_n}\right]$ using Lemma 9.1 to get the result. □

A few technical considerations are necessary, like the error of approximating the Riemann sum by its integral. We do not go into these details for lack of space, but stress the main idea: the error *decreases* as $1/N^2$, *increases* as $n^{0.75}$, and there is a proportionality constant C depending on the regularity of the data. If the data looks close to linear for large stretches, $|F''(z)|^{2/5}$ and its integral should be small, making C small.

ACKNOWLEDGMENTS

This work was partially funded by the National Agency for Research and Development (ANID) / Scholarship Program / DOCTORADO BECAS CHILE/2023 - 72230222.

REFERENCES

[1] Ajay Acharya and Nandini S. Sidnal. 2016. High Frequency Trading with Complex Event Processing. In *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*. 39–42. https://doi.org/10.1109/HiPCW.2016.014

[2] Hirotugu Akaike. 1974. A new look at the statistical model identification. *IEEE transactions on automatic control* 19, 6 (1974), 716–723.

[3] Daniel Berjón, Guillermo Gallego, Carlos Cuevas, Francisco Morán, and Narciso García. 2015. Optimal piecewise linear function approximation for GPU-based applications. *IEEE transactions on cybernetics* 46, 11 (2015), 2584–2595.

[4] bingmann. 2019. *STX-BTree.* https://github.com/bingmann/stx-btree Last Accessed: 2023-12-28.

[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: a height optimized Trie index for main-memory database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 521–534.

[6] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 1789–1792.

[7] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *OSDI*. 155–171.

[8] Herbert A David and Haikady N Nagaraja. 2004. *Order statistics*. John Wiley & Sons.

[9] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (Portland, OR, USA). 969–984.

[10] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree-Are Hybrid Physical Designs Important?. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 177–190.

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

[12] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1189–1206.

[13] Goetz Graefe et al. 2011. Modern B-tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[14] gvinciguerra. 2023. *PGM-Index*. https://github.com/gvinciguerra/PGM-index Last Accessed: 2023-12-28.

[15] Ali Hadian, Behzad Ghaffari, Taiyi Wang, and Thomas Heinis. 2021. COAX: Correlation-Aware Indexing on Multidimensional Data with Soft Functional Dependencies. *arXiv preprint arXiv:2006.16393* (2021).

[16] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

[17] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures. In *Proceedings of the International VLDB Workshop on Applied AI for Database Systems and Applications (AIDB)*.

[18] Ali Hadian and Thomas Heinis. 2021. Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

[19] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *Proceedings of the International VLDB Workshop on Applied AI for Database Systems and Applications (AIDB)*.

[20] Intel. 1997. *Read Time-Stamp Counter*. https://c9x.me/x86/html/file_module_x86_id_278.html Last Accessed: 2023-10-14.

[21] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the International SIGMOD Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*.

[23] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 489–504.

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49.

[25] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 302–313.

[26] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[27] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the International Conference on Management of Data*.

[28] Anisa Llavesh, Utku Sirin, Robert West, and Anastasia Ailamaki. 2019. Accelerating B+tree Search by Using Simple Machine Learning Techniques. In *Proceedings of the 1st International VLDB Workshop on Applied AI for Database Systems and Applications (AIDB)*.

[29] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment (PVLDB)* 14, 1 (2020), 1–13.

[30] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2789–2792.

[31] microsoft. 2022. *ALEX*. https://github.com/microsoft/ALEX Last Accessed: 2023-12-28.

[32] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[33] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The Case for Learned Spatial Indexes. In *Proceedings of the International VLDB Workshop on Applied AI for Database Systems and Applications (AIDB)*.
[34] Naufal Fikri Setiawan, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2020. Function Interpolation for Learned Index Structures. In *ADC*.
[35] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-based Stream Join on a Multicore CPU. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, New York, NY, USA, 2523–2537.
[36] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-sensitive skip list: Efficient range queries on modern cpus. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. Springer, 1–17.
[37] Hari Subramoni, Fabrizio Petrini, Virat Agarwal, and Davide Pasetto. 2010. Streaming, low-latency communication in on-line trading systems. In *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8.
[38] Chuzhe Tang. 2022. *FINEdex*. https://github.com/iotlpf/FINEdex Last Accessed: 2023-12-28.
[39] Chuzhe Tang. 2022. *XIndex*. https://ipads.se.sjtu.edu.cn:1312/opensource/xindex Last Accessed: 2023-12-28.
[40] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.
[41] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520* (2021).
[42] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. 2017. Parallelizing skip lists for in-memory multi-core database systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. IEEE, 119–122.
[43] Yu Ya-xin, Yang Xing-hua, Yu Ge, and Wu Shan-shan. 2006. An Indexed Non-equijoin Algorithm Based on Sliding Windows over Data Streams. *Wuhan University Journal of Natural Sciences* 11, 1 (2006), 294–298.
[44] Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. 2023. FLIRT: A Fast Learned Index for Rolling Time frames. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 234–246.
[45] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-Based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2679–2691. https://doi.org/10.14778/3551793.3551823
[46] Shuhao Zhang, Yancan Mao, Jiong He, Philipp M Grulich, Steffen Zeuch, Bingsheng He, Richard TB Ma, and Volker Markl. 2021. Parallelizing Intra-Window Join on Multicores: An Experimental Study. In *Proceedings of the 2021 International Conference on Management of Data*. 2089–2101.
[47] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *USENIX Conference on File and Storage Technologies (FAST)*. 51–64.

Received July 2023; revised October 2023; accepted November 2023