FLIRT: A Fast Learned Index for Rolling Time frames

Guang Yang*
Imperial College London
London, United Kingdom
guang.yang15@imperial.ac.uk

Liang Liang*
Imperial College London
London, United Kingdom
liang.liang20@imperial.ac.uk

Ali Hadian
Imperial College London
London, United Kingdom
hadian@imperial.ac.uk

Thomas Heinis
Imperial College London
London, United Kingdom
t.heinis@imperial.ac.uk

ABSTRACT

Efficiently managing and querying sliding windows is a key component in stream processing systems. Conventional index structures such as the B+Tree are not efficient for handling a stream of time-series data, where the data is very dynamic, and the indexes must be updated on a continuous basis. Stream processing structures such as queues can accommodate large volumes of updates (enqueue and dequeue); however, they are not efficient for fast retrieval.

This paper proposes FLIRT, a parameter-free index structure that manages a sliding window over a high-velocity stream of data and simultaneously supports efficient range queries on the sliding window. FLIRT uses learned indexing to reduce the lookup time. This is enabled by organising the incoming stream of time-series data into linearly predictable segments, allowing fast queue operations such as enqueue, dequeue, and search. We further boost the search performance by introducing two multithreaded versions of FLIRT for different query workloads. Experimental results show up to 7× speedup over conventional indexes, 8× speedup over queues, and up to 109× speedup over learned indexes.

1 INTRODUCTION

Efficiently managing high volumes of streaming data is essential to large-scale stream processing applications. Streaming data is inherently time-ordered [16] and has a virtually unlimited size; therefore, one can only keep a limited history of the streaming data at a time, called a sliding window. The capacity of a sliding window is defined in terms of a maximum number of records or a time-based limit where records expire from the window after a certain period, e.g., one hour. Managing a sliding window of a high-velocity data stream requires efficient in-memory index structures that support search operations, including range and point lookup, to perform analysis. Such systems are heavily used in high-frequency trading algorithms [3, 46], and spatiotemporal continuous queries [14].

Recently, learned indexing structures have been suggested as efficient alternatives to classical indexes. In learned indexes, a machine learning model replaces the algorithmic components of classic index structures, resulting in a considerable reduction in index size and a significant performance gain. This reduces the memory footprint of the index, thereby boosting the performance. Learned models are particularly efficient for indexing static data; in fact, a significant fraction of the existing learned indexes are read-only. Even though some learned indexes support updates [12, 20, 41], these indexes are not primarily designed for update-heavy workloads such as data streams. This is mainly due to the overhead of training and updating the sophisticated hierarchical models they use to ensure good search performance. Therefore, these indexes are mainly considered search efficient data structures that "allow" for updates similar to traditional B+Trees (see Figure 1).

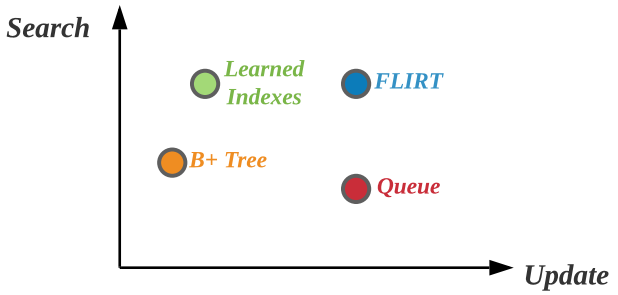

Figure 1: Theoretical search vs update performance trade-off for sliding window scenarios.

Therefore, indexing a continually changing set of keys where the distribution of the keys constantly drift over time is a challenge. Additionally, processing data streams involves tackling the big "velocity" of the data streams, which has not yet been tackled by existing learned indexes.

A more update-friendly solution is to implement the sliding window as a queue (backed by an array or linked list). While a queue supports fast enqueue and dequeue, a range/point lookup within the window requires a binary search or memory-intensive scans. This is shown to be very slow compared to index structures [26, 36], especially with ever-growing sizes of stream data. Therefore, a traditional queue can be seen as an update efficient data structure that supports search operations, which is on the other side of the spectrum compared to typical B+Trees and updatable learned indexes. We seek to combine the update performance of queues with learned models to boost the search performance. Our goal is to create an index structure that is efficient for read-heavy and write-heavy workloads. This is illustrated in Figure 1.

This paper proposes FLIRT, a Fast Learned Index for Rolling Time-frames. FLIRT is specifically designed for the unique pattern of sliding or rolling windows where the keys arrive in a

*Both authors contributed equally to this research.

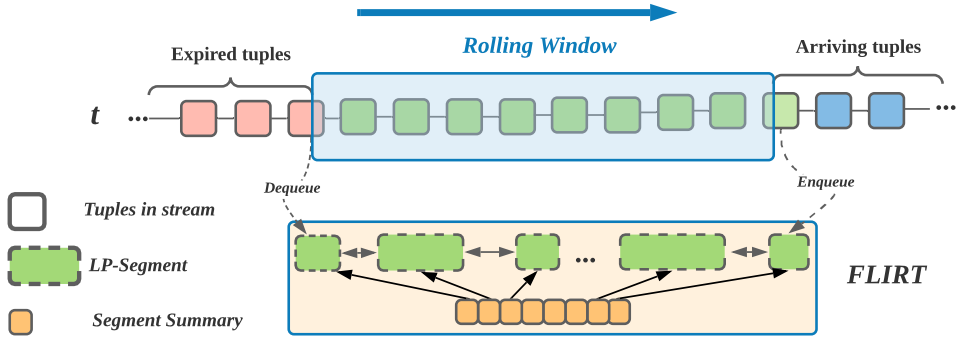

Figure 2: FLIRT Overview

monotonic order (e.g., timestamps) and have a high rate of insertion and deletion as the window slides over the stream. FLIRT is parameter-free and incorporates learned indexing techniques such that the index does not need any re-balancing or re-training when it is fed by the stream.

The contributions of FLIRT are as follows: 1) FLIRT organises streaming records in linearly predicable segments (LP-Segments) and accelerates lookups by allowing records to be located using a linear model. It efficiently supports insertions and deletions of records without re-training or re-balancing. FLIRT uses a circular queue called SummaryList to organise the LP-Segments. While LP-Segments improve the search performance, the queue structure improves the update performance. 2) FLIRT auto-tunes itself by minimizing the search cost. Auto-tuning is crucial for continuous stream processing as the distribution of the data is unknown and ever-changing. Specifically, FLIRT optimises the error threshold to find the perfect balance between the size of the SummaryList and the search range in each LP-Segment. 3) ParallelPartitionedFlirt (PPFlirt) and ParallelSharedFlirt (PSFlirt) are introduced to improve the search performance through multi-threading. Both PPFlirt and PSFlirt use an asynchronous search that scales linearly with the number of threads. PPFlirt partitions the data to reduce the search cost in each thread for balanced query workloads. In PSFlirt, the queries are distributed equally across all threads, which improves performance for queries with a skewed workload.

Results show that FLIRT achieves up to 13× speedup over B+Trees, up to 7× speedup over queues, up to 10^5× speedup over LIST (a streaming index), up to 5× speedup over PGM and up to 109× speedup over ALEX (two efficient and updatable learned indexes). Furthermore, PPFlirt is able to scale up the search throughput of FLIRT by 32×, and PSFlirt improves the search throughput by 4× over FLIRT on a 24-core machine.

2 RELATED WORK

Related work can be classified into index structures for range queries, learned indexes, indexes for sliding windows and stream processing systems.

Index structures for range queries include B+Trees and skiplists. In particular, B+Tree and its extensions [17, 29, 30] are the de-facto standard in most database systems [11].

An alternative approach to B+Trees and other algorithmic indexes is to use machine learning to model the data distribution. For range queries, a learned index model captures the Cumulative Distribution Function (CDF) of the indexed key.

Table 1: Notations

Notation	Definition
Err	Approximation error threshold
W	Window size
N	Number of segments
thd	Number of threads
$\|Q\|$	Number of queries
θ	Degree of skewness
$tune\,rate$	Tune rate
key	Lookup key
seg_i	Segment i
S_i	Slope of segment i w.r.t. $k_{i,0}$
$S_i(key)$	Slope of segment i w.r.t $k_{i,0}$ after inserting key
S_i^u	Upper slope of segment i with respect to $k_{i,0}$
S_i^l	Lower slope of segment i with respect to $k_{i,0}$
D_i	Number of deleted keys in segment i
K_i	Keys stored in segment i
$\|K_i\|$	Number of keys stored in segment i
$k_{i,j}$	Key stored in segment i at index j
$f_{i,j}$	Flag of key stored in segment i at index j
$pos_i(key)$	Predicted position of key in segment i

The learned CDF model can then be used to predict the physical location of the records, given the value of the key. Some learned index frameworks such as RMI support linear and non-linear models [28, 37]. Nonetheless, further experimental results [26, 36] revealed that simple models like linear splines are effective for most real-world datasets. In this regard, splines of linear models are widely adopted by most learned indexes, including FITing-tree [13], Piecewise Geometric Model index (PGM-index) [12], ALEX [8], Radix-Spline [27], Model-Assisted B-tree (MAB-tree) [21], Interpolation-friendly B-tree (IF-Btree) [20] and others [19, 22, 33, 43]. Spline-based indexes group the keys belonging to each spline into a segment and manage the segments in a higher-level index structure. The primary distinction between different indexes is how they train the data (build the splines) and their choice of the higher-level structure that manages the splines.

Some learned indexes are specifically designed to handle updates, including ALEX [8], PGM-index [12], Bourbon [6] and REMIX [51]. Recently, XIndex [47] and FINEdex [31] introduced multi-threading into learned indexes. Nonetheless, there are no learned indexes optimized for sliding window stream processing.

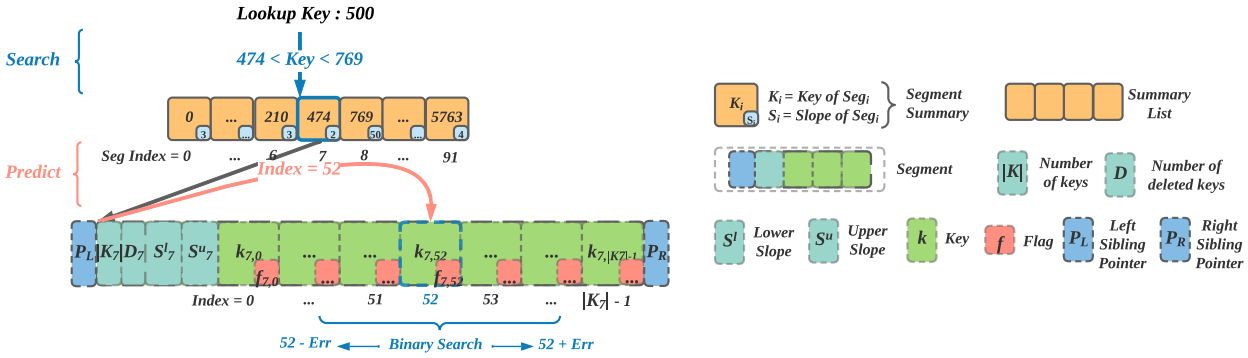

Figure 3: Detailed structure of FLIRT and example search using FLIRT.

Learned indexes have also been extended to spatial and multidimensional indexes [10, 23, 32, 38, 40, 42], indexing data with correlated attributes [9, 18, 39, 48], write-optimised indexes [6, 51], and index recommendation [7, 41].

Machine learning has also been adopted in some components of stream processing systems, especially for frequency estimation [2, 4, 24, 25]. While ML is not yet successfully applied to indexing sliding windows, developing stream indexes is an active area of research and index structures are developed for different applications. Research on streaming systems has shown that using an index improves the performance of stream join [44, 49]. [15] uses a circular array of basic windows to index the streams. The basic windows stores a summary of the keys to decrease memory usage and improve query performance. The data structure of the basic window itself depends on the type of query. The authors state that a linked list basic window (LIST) works well for most cases due to its simplicity. Sliding window indexes have also been extended to the spatio-temporal domain. The Sliding Window Spatio-Temporal index (SWST) [45] and Trails-tree [34, 35] enable disk-based indexing of a limited sliding window over spatio-temporal data.

3 FLIRT OVERVIEW

Indexing data stream deals with continuously generated data whose distribution changes over time. Search efficient data structures, such as the B+Trees and learned indexes [8, 12, 13], support updates. However, their performance deteriorates with high-velocity insertions and deletions. Furthermore, these indexes are inefficient for sliding-window update patterns, where insertion occurs at the end (enqueue), and deletion occurs at the front (dequeue). Update-efficient data structures such as queues are very efficient at updating; however, queues are not designed for fast searches. A detailed breakdown is shown in Section 6.

FLIRT is tuned for fast searching and updating on sliding-window data streams. Figure 2 shows a time series streaming data where the keys are timestamps of a dataset. As time passes, new timestamps are inserted at the end of the index, and the expired timestamp is removed from the start of the index. The remainder of the paper refers to FLIRT insertions and deletions as enqueues and dequeues, respectively.

Learned indexes consider indexes to be functions that map a key to a physical storage location. They then use a machine learning model to approximate the function [28]. FLIRT approximates the mapping using piecewise linear functions, an approach similar to spline-based learned indexes [8, 12, 13, 27, 33]. Spline-based learned indexes have shown to be inexpensive to compute and store. Most importantly, they are update-efficient. We refer to this mapping as linearly predictable segments (LP-Segments).

The key insight for learned indexes is that they provide an approximation since the predicted location may not be the actual location of the key. To control the precision of the approximation, learned indexes use an error threshold, which determines the maximum distance between the actual and predicted positions. The optimal error threshold depends on the distribution of the dataset. FLIRT auto-tunes the error based on the search cost, making it parameter-free, and is discussed in detail in Section 4.

Figure 2 shows an overview of FLIRT. Each LP-Segment, in green, holds a linear regression model. Each segment contains a set of metadata, a set of keys covered by the regression model, and a set of payloads. The SummaryList queue stores a summary of the segments (in orange). The summary, which contains metadata of each segment, allows FLIRT to manage each segment for cache-efficient lookups. Segments are connected to their neighbours to efficiently execute range queries with high selectivity.

FLIRT's data structure is suited for asynchronous parallel query processing. We introduce two multithreaded versions of FLIRT, called ParallelPartitionedFlirt (PPFlirt) and ParallelSharedFlirt (PSFlirt), to further improve the search performance. PPFlirt improves performance by partitioning the data to reduce the data searched in each thread. Each thread has a local FLIRT and executes searches asynchronously without communication between threads. An update thread manages the enqueue and dequeue operations. PSFlirt keeps the data in one global FLIRT and does not partition the data. The search threads access the non-updating segments freely, while the update thread enqueues and dequeues from the back and front segments. In both versions, increasing the number of threads will only increase the number of search threads due to the nature of sliding-window data streams. Details are discussed in Section 5.

FLIRT aims to achieve the following goals with respect to traditional data structures (B+Tree and queue) and learned indexes (ALEX and PGM-index). (1) A faster search time for larger window sizes. (2) Faster insertions and deletions performance. (3) Reducing the index size to achieve better cache efficiency. PPFlirt and PSFlirt further improve the search performance while maintaining a high update performance. Table 1 summarises the notations used in this paper.

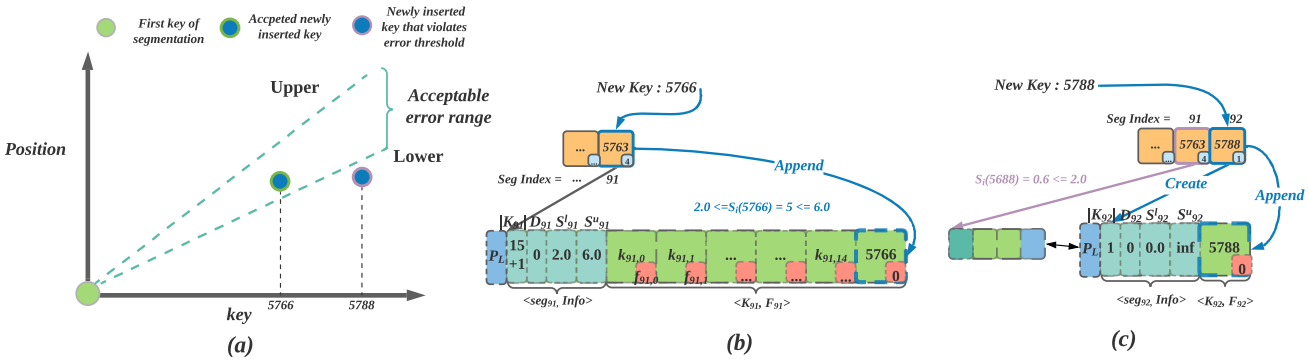

Figure 4: Example enqueue using FLIRT. (a) Segmentation algorithm used by FLIRT, green point indicates the first key of the segment, and the blue point indicates a newly inserted key. (b) Inserting a key that does not violate error threshold. (c) Inserting a key that requires a split.

4 FLIRT DESIGN

FLIRT manages the LP-Segments using an array-based circular queue structure shown in Figure 3. A circular queue is chosen over a linear queue for its performance advantage and memory efficiency. An array-based circular queue is chosen to reduce memory access and cache misses compared to a linked-list based queue.

We store a segment summary, including the starting key and slope, in the queue to find the predicted position without accessing the segment itself, thereby reducing memory access. The starting key identifies the range of keys each segment covers, and the slope finds the predicted position. The LP-Segments are connected via two sibling pointers to improve the range query performance.

Each LP-Segment contains two error bounds, an upper and a lower error bound, to ensure the keys in each segment are within the error threshold. Initially, when only one key is in the segment, the upper and lower error bounds are infinity and zero, respectively. The slope of upper error bounds starts to decrease when inserting keys while the slope of the lower error bound increases. The two error bounds form a narrowing cone with more keys [13]. The slope of the segment is the average of the two error bounds.

Each key is associated with a deletion flag to allow keys to be deleted without shifting. Shifting keys is expensive and more costly for learned indexes because moving keys requires retraining the segment. We carry out a pseudo-delete by flagging the key and actually deleting the entire segment once all the keys have expired.

4.1 Search

FLIRT first searches the SummaryList to locate the segment that contains the lookup key. Each segment seg_i contains $k_{i,0} \leq K_i < k_{i+1,0}$. The search finds the last segment whose $k_{i,0}$ is equal to or smaller than the lookup key. An example is shown in Figure 3. We find the segment with a starting key of 474 contains the lookup key 500. The process is similar to a non-leaf node B+Tree traversal.

FLIRT uses linear search when the SummaryList is small and binary search when the SummaryList is large. Empirical studies show linear search is more efficient for smaller arrays and binary search is more efficient for larger arrays, and the threshold for switching between the two is 256 bytes [1]. The linear search algorithm is trivial: compare the lookup against each key in the SummaryList until the search condition is met. A binary search algorithm uses an inequality check to satisfy the search condition. For example, if $K[mid] \leq k_{search} < K[mid+1]$, segment seg_{mid} contains the search key, otherwise the binary search continues.

Inside the LP segment, we apply the linear regression model $S_i \times (key - k_{i,0})$ to find the predicted position of the key. In our example in Figure 3, the predicted position is 52. Since the predicted position and the actual position is guaranteed to be no larger than the Err, a local binary search between the error bounds $[pos_i(key) - Err, pos_i(key) + Err]$ finds the lookup key. If we find the key within the error bound, we check its deletion flag $f_{i,j}$ to determine if the key has been removed. If the key is not within the error bound, the key does not exist in the index.

4.2 Enqueue

There are two cases when inserting into the segment: (1) after insertion; the maximum error is less than the error threshold, and (2) after the insertion, the maximum error is greater than the error threshold. We check if the inserted key violates the error threshold by checking if the slope falls within the error bound shown in Figure 4a. The slope for segment i is computed with:

$$S_i(key) = \frac{|K_i|}{key - k_{i,0}} \tag{1}$$

Where, $S_i(key)$ is the slope of segment i after key is inserted, key is the insertion key, $k_{i,0}$ is the first key of seg_i, and $|K_i|$ is the position where key is inserted. $|K_i|$ is actually the number of keys in seg_i, which is same as the position for newly inserted keys when enqueuing.

An example of the first case is shown in Figure 4b. The slope falls within the error bound, and the key is inserted at the end of the segment. We use a dynamically allocated array that has an atomized insertion cost of $O(1)$ to store the keys. The segment updates the slope and error bounds after each insertion. The upper bound and lower bound are computed by adding or subtracting the error bound to Equation 1:

$$S_i^u = \min\left(S_i^u, \frac{|K_i| + Err}{key - k_{i,0}}\right) \tag{2}$$

$$S_i^l = \max\left(S_i^l, \frac{|K_i| - Err}{key - k_{i,0}}\right) \tag{3}$$

The error bounds shrink with more keys. The degree of shrinkage depends on the data distribution. If the distribution is close

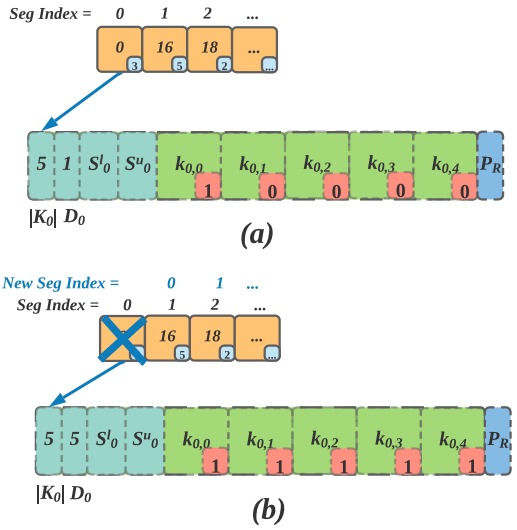

Figure 5: Example dequeue using FLIRT. (a) Segment is not empty after dequeue. (b) Segment is empty after dequeue.

to linear, the error bounds shrink very slowly. When the distribution is less predictable, the error bounds tend to decrease rapidly, and keys are more likely to be outside of the error bound.

An example of the second case is shown in Figure 4c. If the key is inserted into the segment, we cannot guarantee the true position is at a maximum distance of Err away from the predicted position. We split the segment once the error threshold is violated which eliminates the need to retrain keys in the current segment. The inserted key is the first key of the new segment and will be appended to the SummaryList.

4.3 Dequeue

There are two cases when deleting from each segment: (1) After deletion, the segment is not empty. (2) After deletion, the segment is empty. For the first case, shown in Figure 5, we select the first key in the segment $k_{i,0}$ and update its deletion flag $f_{i,0} = 1$. A deletion counter is updated $D_i = 1$. The deletion counter is used as the index for subsequent deletes. The segment is empty once all the keys are deleted $|K_i| == D_i$, shown in Figure 5b. The entire segment is then removed from the SummaryList, and all records within the segment are deleted. The right sibling of the deleted segment becomes the first segment in FLIRT.

The reason to use a deletion flag is to reduce the cost of shifting every time a deletion occurs, hence, reducing latency. Shifting also requires the regression model to be retrained or the error to be increased to accommodate the change in position of keys. Hence, we want to prevent shifting at all costs. In return, the search operation requires an additional check. The experimental evaluation shows that the cost is minimal and does not impact FLIRT's performance.

4.4 Auto-tune

The performance of FLIRT depends on the error threshold Err. The error threshold determines the error bound in each segment and influences the number of segments (determined by window size and error bound). Large error thresholds allow more keys stored in each segment, hence, reducing the number of segments and results in a smaller SummaryList. In the extreme case, all keys are contained in one segment. The search in the SummaryList is improved, but the local search in each segment worsens. Small error thresholds generate smaller segments and larger SummaryList. In the extreme case when the error is zero, each segment stores two keys (the cone will have an area of zero). FLIRT essentially behaves like a queue and gains no benefit from the regression model.

We can determine an optimal error threshold for a static dataset. However, streaming datasets are dynamic, and their distribution is constantly changing; therefore, there is no one value that fits all situations. FLIRT dynamically changes the error threshold using a cost model. The cost model minimizes the combination of the cost of the SummaryList search and the cost of the local search:

$$C = \underbrace{\log(N)}_{\text{SummaryList Search}} + \underbrace{\log(Err)}_{\text{Segment Search}} \quad (4)$$

FLIRT initially uses a default error threshold of $Err = 256$ and bulk loads the keys until the number of keys reaches the window size. The initial cost of the index is $C_{initial} = \log(N) + \log(Err)$, where N is the number of segments. We double the error threshold $Err_{new} = 2Err$ after bulk loading and update the index. A tune rate controls how many updates before we tune the error threshold. By default, the value is 10% of the window size. Once we hit the tune rate, we estimate the cost of the index $C_{current} = \log(N_{new}/tune\,rate) + \log(Err_{new})$, where N_{new} is the number of newly inserted segments. If the current cost $C_{current}$ is lower than the initial cost $C_{initial}$, we continue in the same direction and double the error. For the opposite situation, we reverse the direction and halve the error. This process is similar to the gradient descent algorithm, where we find a direction that minimizes the cost. The current cost replaces the initial cost, and we continue to update the index until we hit the tuning rate.

Similar to having large learning steps in the gradient descent problem, doubling and halving the error threshold may skip over the optimal error. Therefore, we reduce the tuning step when the error threshold bounces between two values using a step decay approach. Specifically, we reduce the tuning step by half once the error threshold bounces between two values 5 times. We also have to consider distribution changes which cause the optimal error to shift. Therefore, we increase the tuning step if the error threshold continuously grows in one direction by reversing the reduction in the tuning step (doubling the tuning step once the error threshold moves in one direction for 5 times).

5 PARALLELIZING FLIRT

ParallelPartitionedFlirt (PPFlirt) and ParallelSharedFlirt (PSFlirt) are two multithreaded versions of FLIRT designed to improve the search throughput. Both versions have a single update thread and an arbitrary number of search threads. This is because data are always enqueued at the end and dequeued from the front. Multiple update threads will lead to race conditions and require considerable synchronisation. PPFlirt partitions the data such that each thread has its own local data and is designed for general use case. While PSFlirt shares the data and allows threads to access different parts of the data and is optimised for skewed workloads.

5.1 Parallel Partitioned Flirt (PPFlirt)

Figure 6a shows the structure of PPFlirt. Data is range partitioned, and each partition has its own local FLIRT. The goal is to reduce

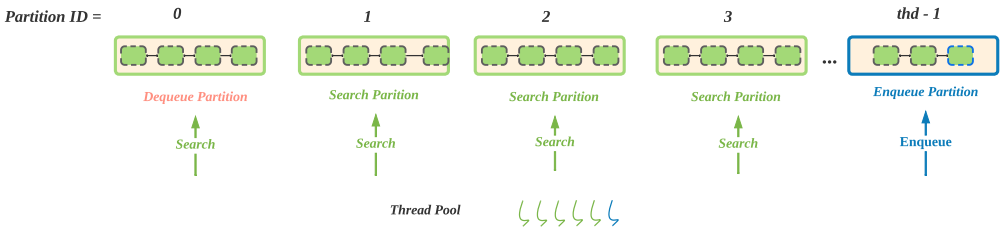

Figure 6: Overview of Parallel Partitioned Flirt (PPFlirt): Each partition is maintained exclusively by its own thread.

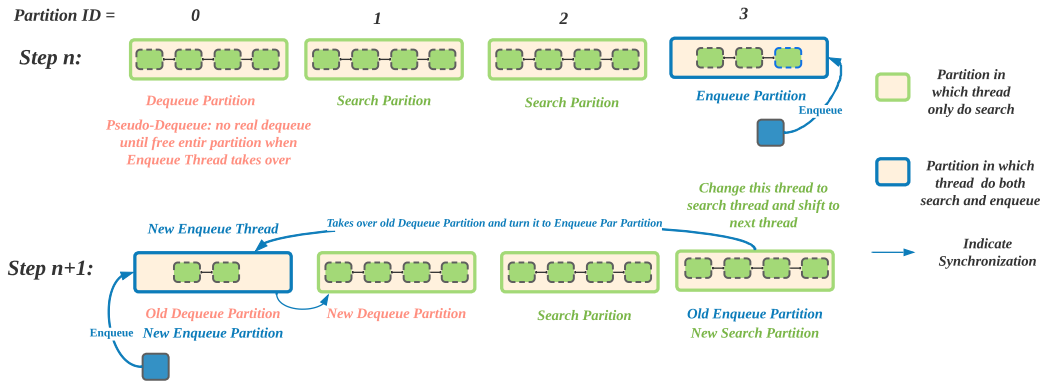

Figure 7: Enqueue and Dequeue in PPFlirt.

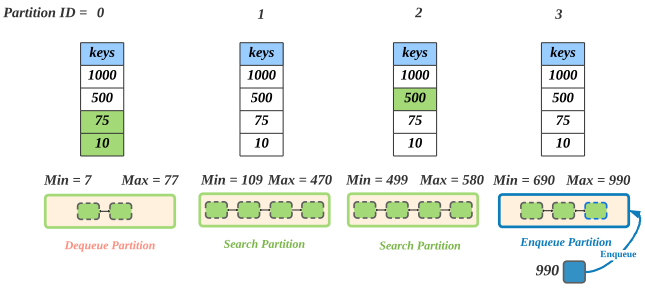

Figure 8: Asynchronous search in PPFlirt.

the amount of data each thread processes, thereby, increasing search throughput. We partition each thread to have an equal number of keys for load balancing. The local capacity of each thread is $\frac{W}{2(thd-1)}$. The -1 ensures that one of the threads is initially empty for enqueuing.

We initialize PPFlirt by partitioning the data into equal sizes with key ranges increasing with thread ID. The dequeue partition is always behind the enqueue partition. Initially, the first partition is the dequeue partition, and the last partition is the enqueue partition as we consider the threads to be in a circular loop. The last partition is assigned to the update thread, while the rest of the partitions are assigned to search threads. Each thread, except for the update thread, builds a local FLIRT for the portion of data they manage and stores the first and last key to keep track of the key range.

5.1.1 Enqueue & Dequeue. Since the window size of the index is constant, an enqueue is paired with a dequeue. However, updating the two partitions requires synchronization, which reduces the performance of the system. Instead, we advocate for a "pseudo" delete, where we keep track of which keys are deleted and do not modify the dequeue partition itself. Since the enqueue and dequeue operation are paired, and each partition is independent, the position to enqueue is equal to the position to dequeue. For example, if we enqueue a key into position 0 of the enqueue partition, the key at position 0 in the dequeue partition is deleted to ensure a constant window size. Therefore, we use a thread-safe update counter to keep track of the enqueue/dequeue position.

The update thread inserts to the enqueue partition, which is initially empty. Internally, it calls the FLIRT enqueue operation and increments the update counter. A search is carried out in the update thread after each enqueue since keys may fall within it's key range. The dequeue partition is managed by a search thread with a special search operation. The search operation finds the position of the lookup key, if it exists, and compares the index with the update counter. The key has expired if the position is less than the counter.

Once the enqueue partition is full, it becomes a search partition, and the dequeue partition becomes the new enqueue partition. The actual deletion occurs during this transfer when the dequeue partition is emptied. In Figure 7, the new enqueue partition moves to thread 0, thread 1 becomes the new dequeue partition and thread 3 becomes a search partition. Thread 0 is the new update thread, and the original data in thread 0 is emptied. The update counter is reset to zero to indicate the new enqueue/dequeue position. No synchronization is needed for this transfer, as all updates occur in one thread.

5.1.2 Search. The order in which threads execute depends on the scheduler and is not deterministic. PPFlirt search threads

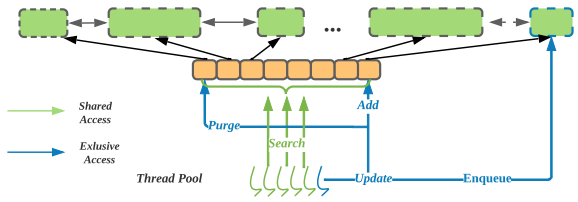

Figure 9: Overview of Parallel Shared Flirt (PSFlirt): All segments are shared among multiple threads, except for the enqueue thread.

work asynchronously to increase throughput. We can use a master thread as a range dispatcher that assigns keys to the corresponding partition. However, this process is sequential and not parallelizable and may create a performance bottleneck. Furthermore, enqueue and dequeue require the range to be updated and require synchronization to prevent race conditions.

Alternatively, each thread will have a search queue with the same lookup keys shown in Figure 8. There is no master thread that dispatches to each search queue; all threads asynchronously read from one queue of queries. Each thread has an independent "view" of the queries, and once all threads surpass a certain part of the queries queue, the old queries can be deleted. To reduce unnecessary searches, each thread first checks if the lookup key is within its range, which takes $O(1)$ time, before carrying out a FLIRT search. No race conditions will occur since each key can only exist in one partition. Section 6.10 shows that the search queue dispatcher performs better than the range dispatcher; however, both are not the bottlenecks of the system.

Range queries may intersect multiple partitions; therefore, the check will consider the search radius. If the key range intersects the lower and upper bound of the range query, the thread will carry out a FLIRT range search. The final result is aggregated from multiple threads in an asynchronous manner. There is a special case where the range query covers an entire partition which occurs when the query range is large or the partition is small. In this case, we traverse the local FLIRT and return all non-expired keys to avoid the overhead of searching.

5.2 Parallel Shared Flirt (PSFlirt)

Figure 9 shows the structure of PSFlirt. It keeps the data in one global index, and threads can freely access different parts of the index asynchronously. PSFlirt is designed for query workload balance such that each thread will process a similar number of queries. It achieves higher search throughput by simply having more search threads. The number of data each thread has to search is the same as FLIRT. Therefore, the throughput for a balanced workload will be lower than PPFlirt because each thread has to process a larger index. However, PSFlirt is more suited for skewed query workloads where the search keys are more concentrated in parts of the range. In the extreme case, one of the threads in PPFlirt will be saturated with queries while the rest do nothing. PSFlirt's threads will have a balanced query load, thereby increasing parallelism.

5.2.1 Enqueue & Dequeue. PSFlirt also uses "pseudo" deletes to reduce synchronization costs. Similar to PPFlirt, we use a thread-safe counter to store the update position. Since PPFlirt uses a single index and the segment sizes vary with key distribution, the enqueue position and dequeue position is not in sync. Each segment needs to keep track of the sequential ID of its first key. The sequential ID is the index of the key in the entire stream and increases with data. For example, assuming a window size of 100, the last key will have a sequential ID of 99. The next key inserted into the index will have an ID of 100, and the subsequent key will have an ID of 101.

For every update, the update thread first enqueues the key to the last segment and increments the update counter before checking if the first segment needs to be deleted. PSFlirt uses two reader-writer locks for synchronization. The first lock is located at segment level and is used for enqueues that does not split the segment. Only readers who need access to the last segment will have to wait. The second lock is at the queue level and is only triggered when a segment is appended to the SummaryList. All readers must wait in this case. We check whether to delete the first segment by checking if the update counter is equal to the sequential ID of the second segment. If the sequential ID is equal to the counter, the queue level lock is triggered, and the first segment is removed from the SummaryList.

Since the sequential ID and update position grows indefinitely with more data, there is a chance of overflowing. Hence, we reset these two values to zero if the next update triggers an overflow. PSFlirt then scans through the SegmentList and updates the sequential ID of the first key in each segment.

5.2.2 Search. Each search thread will have its own search queue with different lookup keys and uses a reader lock to freely traverse PSFLirt until an update is triggered. All threads can access the first segment; therefore the search operation needs to check if the key has expired. The sequential ID of a key in seg_i is the position of the key in seg_i plus the sequential ID of the first key $k_{i,0}$. The key has expired if the sequential ID is less than the update counter. The search operations, including range search, are the same as FLIRT with the addition of reader locks for synchronization.

6 EXPERIMENTAL EVALUATION

This section evaluates the performance of FLIRT, PPFlirt and PSFlirt. We first show an analysis of the auto-tune method and the performance of FLIRT against different baselines. We then show the performance of FLIRT under varying window sizes, read-write ratios, number of operations and query ranges. The performance of PPFlirt and PSFlirt is compared with FLIRT for different numbers of threads. The highlights from our evaluation are:

- FLIRT's auto-tune method is able to find the best performing error threshold.
- Compared to traditional indexes, FLIRT achieves up to a 6.9× speed-up against a B+Tree and up to an 8.0× speed-up over a circular queue.
- Compared to updatable learned indexes, FLIRT achieves up to a 108.5× speed-up over ALEX and up to 5.0× speed-up over PGM.
- Compared to the streaming index, FLIRT achieves up to a 10^5× speed-up over LIST.
- PPFlirt and PSFlirt achieves up to 32.0× and 4.1× increase in search throughput over FLIRT. PSFlirt has consistent performance under skewed workloads.

6.1 Experimental Setup

FLIRT and ParallelFLIRT are implemented in C++ and the experiments were conducted with gcc on a machine with Intel E5-2680

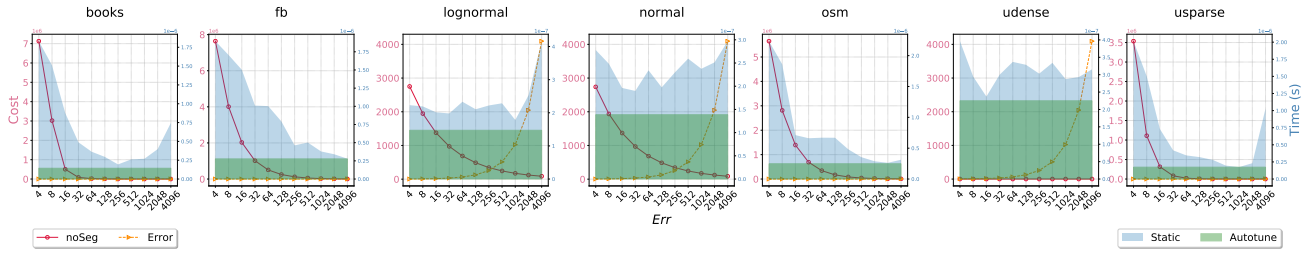

Figure 10: Performance comparison of FLIRT with auto-tuned *Err* and FLIRT with a predefined static *Err*

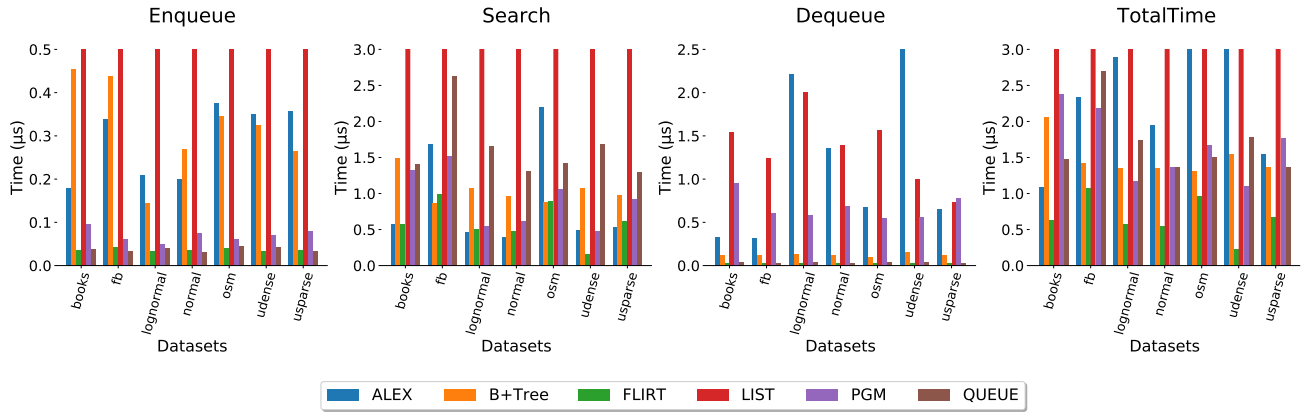

Figure 11: Overall performance breakdown of FLIRT against baselines

v3@2.50GHz running Centos 8. We use 7 datasets from the SOSD benchmark [26] to ensure that our results are reproducible and comparable with other learned indexes. Specifically, 3 real-world datasets (*books*, *fb* and *osm*), and 4 synthetic datasets (*lognormal*, *normal*, *uniform dense* and *uniform sparse*) were used for the experiments. We are aware of the benchmark datasets used for streaming such as YSB [5] and Stock [50]. However, these datasets are used to showcase how streaming systems schedule the data, such as arrival/departure rates. For learned indexes, however, we are more concerned about how indexes perform with different distributions of the data.

The baselines include a STX implementation of B+Tree[1], a circular queue using arrays, LIST using linked list as the basic window, PGM-index[2] and ALEX[3]. Bourbon and REMIX are insert-optimised and do not describe how deletions work for a learned LSM-tree. LSM-trees itself does not support efficient deletions. Hence, Bourbon and REMIX are not included in the baselines.

The number of records in the index is determined by the window size and is kept constant with updates to simulate a tuple based sliding window. The default window size is $100M$ keys such that the data does not fit in the cache. Learned indexes are pointless if the data fits in cache, as it would be difficult to outperform a B+Tree.

The default workload involves a single update (enqueue and dequeue) followed by a search. Each search consists of finding the $rank(x)$, where x is the lookup key [12]. The combination of an update and search is considered an operation as it represents a sliding window moving one step. The default number of operations is $1M$ and the tune rate is 0.1% of the window size.

[1]https://github.com/bingmann/stx-btree
[2]https://github.com/gvinciguerra/PGM-index
[3]https://github.com/microsoft/ALEX

We measure the performance using execution time and throughput. Execution time is the average execution of each operation (enqueue, dequeue, and search) and is timed using the C++ standard `chrono` library. Throughput measures the executions per second and is used for the multi-threaded cases.

For skewed query workloads, we use a Zipfian distribution with different degrees of skewness θ from 0.1 to 0.9. The number of queries Q in each process i is given by:

$$|Q_i| = \frac{|Q|}{i^\theta \times \sum_{j=1}^{thd} \frac{1}{j^\theta}} \tag{5}$$

Where, $|Q|$ is the number of queries and thd is the number of threads (set to 12). For PPFlirt, $|Q_i|$ is the number of queries in each thread. More queries are given to the first threads as θ increases. For PSFlirt, $|Q_i|$ is the number of keys in each different part of the data. Similarly, more data is sampled from the front of the data as $theta$ increases. We then distribute the skewed queries evenly across search threads. We test the search performance under a skewed workload; hence, no updates are performed.

6.2 Evaluation of Auto-tune Method

As discussed in Section 4.4, the performance of FLIRT depends on the error threshold. We first want to verify that there is a minimum error in the cost model to justify its use. The red and orange lines in Figure 10 correspond to the left y-axis and shows the SummaryList search cost ($\log(N)$ in red) and Segment search cost ($\log(Err)$ in orange) against error values from 4 to 4096. We can clearly see the intersection between the two costs for *lognormal* and *normal*; therefore, an error that minimises the cost does exist. We are unable to see the intersection for the rest of the datasets due to the number of segments being too large. The

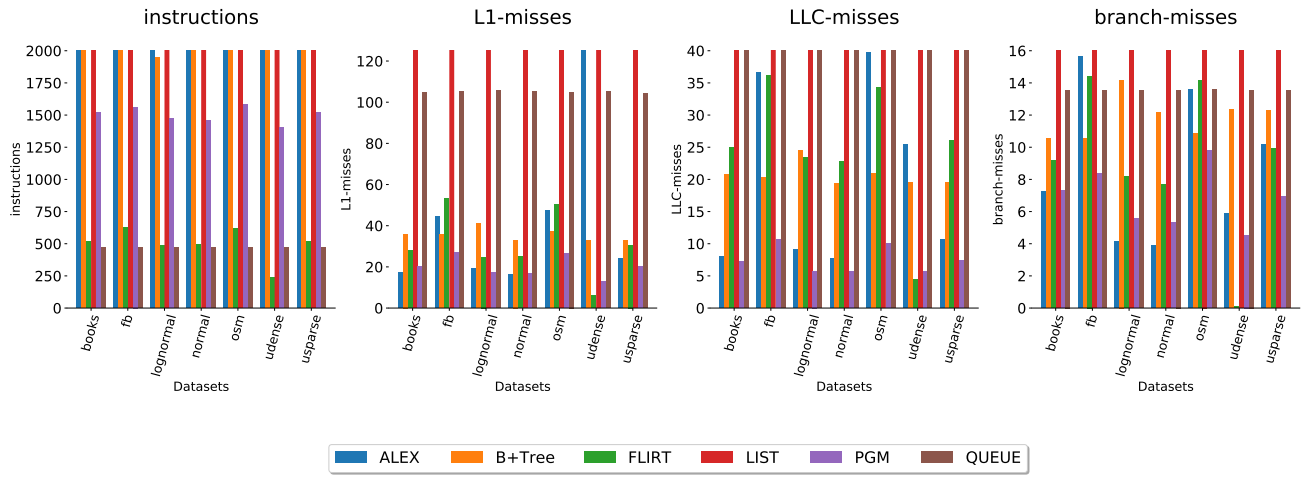

Figure 12: Analysing the processor-level performance counters

intersection does exist and can be seen when we decrease the window size. Additionally, we observe in our experiments that larger window sizes tend to prefer larger error thresholds.

We then verify whether the cost model describes the behaviour of FLIRT and verify the effectiveness of the auto-tune method. The blue and green areas in Figure 10 correspond to the right y-axis and shows the execution time of FLIRT without auto-tuning (blue) for different errors and execution time of FLIRT with auto-tuning (green). FLIRT without auto-tunes uses a static error threshold given by the value in the x-axis. FLIRT with auto-tuning does not depend on the x-axis, hence the result is constant with varying errors. The results highlight two key observations. (1) The cost model can be verified by FLIRT without auto-tuning because the minimum execution time and minimum cost have a similar error threshold. (2) The auto-tuning method is able to find the error with the lowest execution time.

6.3 Comparison with the baselines

We compare FLIRT against five baseline indexes (B+Tree, Queue, PGM-index, ALEX and LIST). The branching factor for the B+Tree is varied from 4 to 4096, and the branching factor that results in the lowest total execution time is shown. A similar process is applied to the PGM-index, which uses an error parameter. The circular queue, ALEX and LIST are parameter-free. Figure 11 shows the execution time against different datasets for different operations. The total time is the combination of the enqueue, dequeue and search time. FLIRT is shown to outperform the baseline across all datasets.

One reason for FLIRT's superior performance is its small size. The index size of FLIRT is 0.04% of the window size and is two orders of magnitude less than rival learned indexes, which take up 5.5 – 14% of the window size. The execution time is dominated by the search time and can be seen by comparing the total time and search time figures. Update performance is quite consistent across datasets and is on average 10× faster than the search time.

For the enqueue performance of the baselines, queue achieves the best performance, and LIST has the worst performance, followed by the B+Tree. We did not expect LIST to have the worst performance since it is based on a linked list. The main inefficiency comes from the C++ doubly linked list being very cache inefficient and requires considerable memory especially when the index size (window size) is large (Figure 13). This further emphasises why cache efficiency is important. However, LIST performs consistently across baselines which verifies the authors' claim in [15]. PGM-index has the best performance regarding learned indexes due to it having a similar piecewise linear approximation as FLIRT. The overhead comes from the hierarchical model requiring multiple levels to be trained. ALEX suffers under a sliding window workload. We think this may be due to the overhead from the more sophisticated hierarchical structure optimised for random insertions.

Queues are expected to have the best dequeue performance for the baseline. The learned indexes are not efficient for dequeueing and are outperformed by the B+Tree. ALEX is especially inefficient for data with a uniform distribution in the micro-level (*lognormal*, *normal*, *uniform dense*). We suspect this is because ALEX puts a lot of keys in a single segment since the micro-level distribution is linear. The main takeaway is that FLIRT is able to match the update performance of queues while "learning".

In terms of search performance, learned indexes perform the best as expected. B+Tree has an average performance, while queue and LIST have the worst performance. LIST is expected to have the worst performance since it uses a linked list, which does not allow for binary search. The rest of the indexes abide by their typical behaviour. Tree-like structures have significant overhead when updating; however, the gain in search performance over linear structures is not nearly as much as one would expect. FLIRT combines the update performance from linear structures and search performance from learned indexes to outperform the baselines.

6.4 CPU-level Micro-Analysis

We use `PerfEvent`[4] to give insight the performance of each index. Figure 12 shows the L1-cache misses, last level cache misses, branch misses and the number of retired instructions against different datasets.

The results show that FLIRT has a low instruction count, similar to a queue. The L1 cache miss is similar to the learned indexes, but the LLC misses are higher than the learned indexes. We suspect this is due to the use of a queue like structure rather than a tree. Interestingly, the branch misses of FLIRT seem to correlate with the linearity of the distribution. *osm* is shown to have the

[4]https://github.com/viktorleis/perfevent

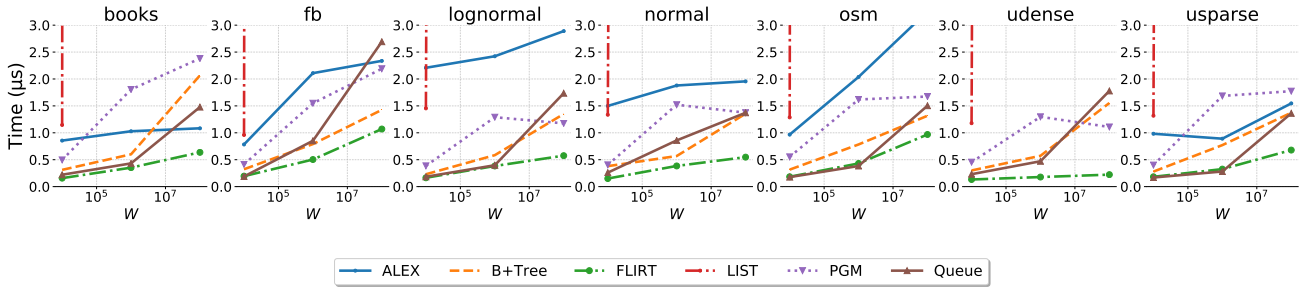

Figure 13: Execution time (μs) of all methods with varying window sizes

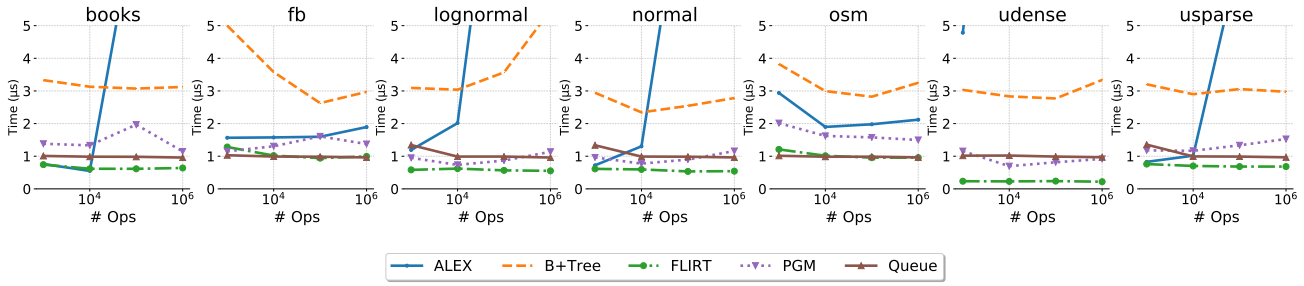

Figure 14: Execution time (μs) of all methods with varying number of operations.

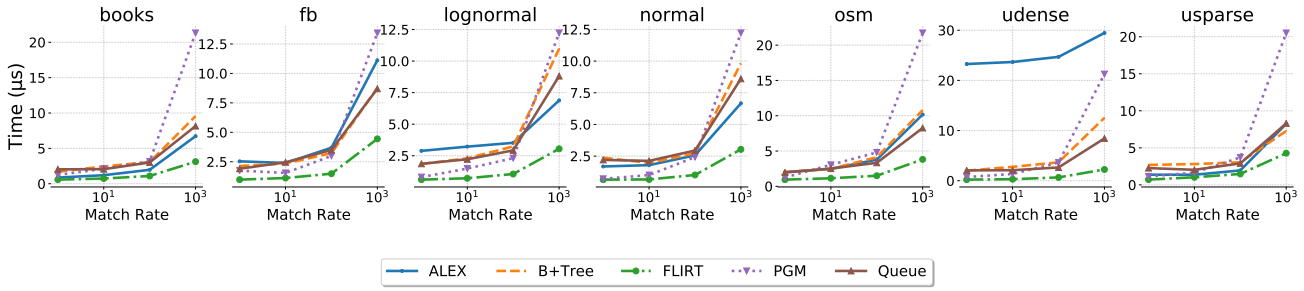

Figure 15: Execution time (μs) under different match rates.

highest branch misses while *uniform dense* is shown to have close to zero branch misses.

The queue has a high cache misses rate due to its linear structure and the need to access different parts of the queue during each operation since the entire queue cannot fit in the cache. B+Tree has an average performance across all performance counters, which is expected for one of the most generic indexes. It also explains why it has the most average performance in terms of execution time. The PGM-index is shown to be the most cache efficient due to its use of linear piecewise approximations and a tree-like structure. However, the results do not explain why the PGM-index is inefficient when dequeueing. ALEX is also cache efficient for most datasets, which suggests that learned indexes are generally cache efficient. The cache miss rate of ALEX is related to enqueue and search performance, and the instruction count relates to the dequeue performance. The combination of a queue structure and a linked list makes LIST inefficient for all performance counters, which also explains its' poor performance.

6.5 Window Size

The window sizes are varied from $10K$ to $100M$ to compare the performance at different data sizes, and the results are in Figure 13. FLIRT maintains the best performance under all window sizes, with the execution time linearly increasing with window size. For small window sizes, all indexes perform well as they can all fit in the cache.

As for the baselines, B+Tree and queue show a two-stage increase where the rate of increase is lower while the window size is under $1M$. We suspect that $1M$ records is the point when the data cannot fit in cache, and performance of B+ tree and queues begin to suffer. The PGM-index becomes more efficient after $1M$, which shows the benefits of learned indexes. Both B+Tree and PGM can achieve better performance by tuning the parameter (branching factor and error) for each window size. ALEX is very dependent on the data distribution. LIST performs poorly with large windows as the linked list gets larger. FLIRT, on the other hand, is parameter-free and can adapt to different window sizes.

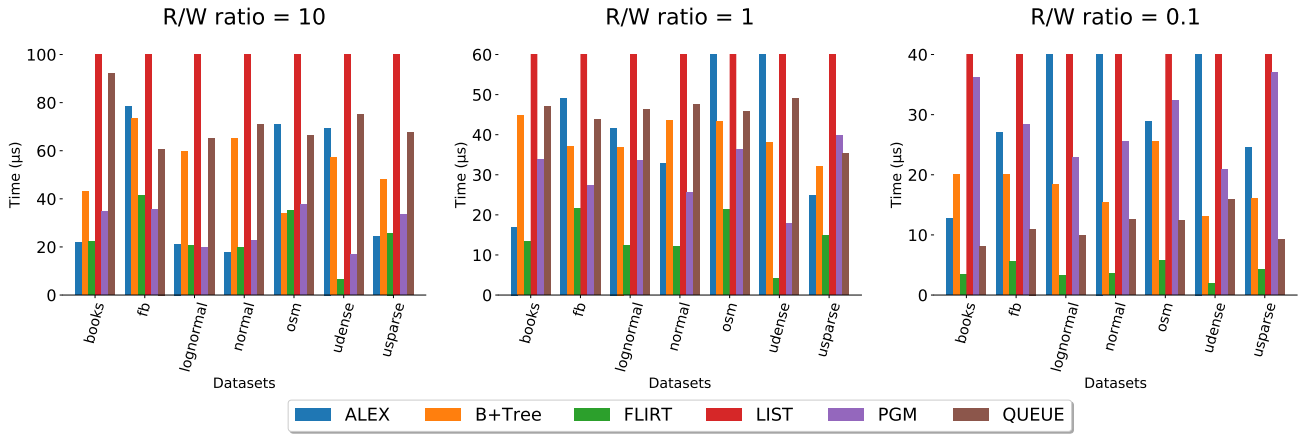

Figure 16: Execution time (μs) under different read-to-write workloads.

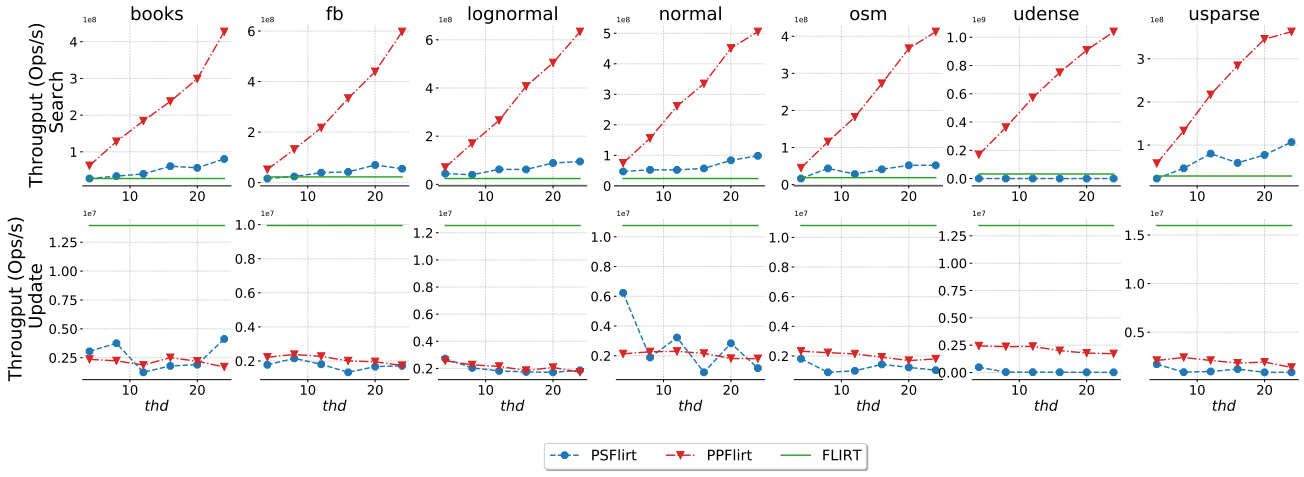

Figure 17: Comparison between the search and update performance of FLIRT with PPFlirt and PSFlirt in terms of throughput.

6.6 Number of Operations

We vary the number of operations from $1K$ to $1M$, while keeping the window size at $100M$, to test the indexes under continuous updates. One operation is a combination of one enqueue, dequeue and search. This also leads to the macro-level distribution changes in the sliding window, especially for *osm*. The results is presented in Figure 14. LIST is not shown due to it taking on average $2.5 \times 10^4 \mu s$. Most of the indexes perform consistently (including LIST) and are resilient against changes in the macro-level distribution. The only exception is ALEX.

6.7 Range Query

We evaluate the performance of range queries using a match rate from 1 to 1000. The match rate is the scan length and determines how many items to scan in each range query. The results are shown in Figure 15. LIST is not shown due to it due it taking on average $3.2 \times 10^4 \mu s$. FLIRT outperforms the baselines across all match rates. The execution times increase for all indexes as the match rate increases, and the performance decreases significantly once the match rate is over 100.

6.8 Read/Write Ratio

We evaluate the performance of the indexes under different workloads by varying the read to write ratio from 10:1 to 1:10. The results are shown in Figure 16. The result complies with Figure 1. Tree-based learned indexes and B+Trees are shown to perform better with read-heavy workloads, and queue type structures are more suited for write-heavy workloads. FLIRT benefits from having LP segments in a read-heavy workload and achieves similar performance to PGM and B+Tree. FLIRT benefits from having a queue like structure in a write-heavy workload allowing it to achieve the best performance. The performance of FLIRT decreases with a higher read to write ratio (y-axis), and we use multithreading to improve the search performance.

6.9 Parallel FLIRT Performance

The performance of PPFlirt and PSFlirt is compared against FLIRT. The window size and the number of updates are $50M$, and the number of threads is varied from 4 to 24. Since the system performs updates and searches in parallel, we track the time it takes to complete $50M$ updates as well as the number of searches in that duration. The update throughput is the number of operations over the duration, and the search throughput is the number

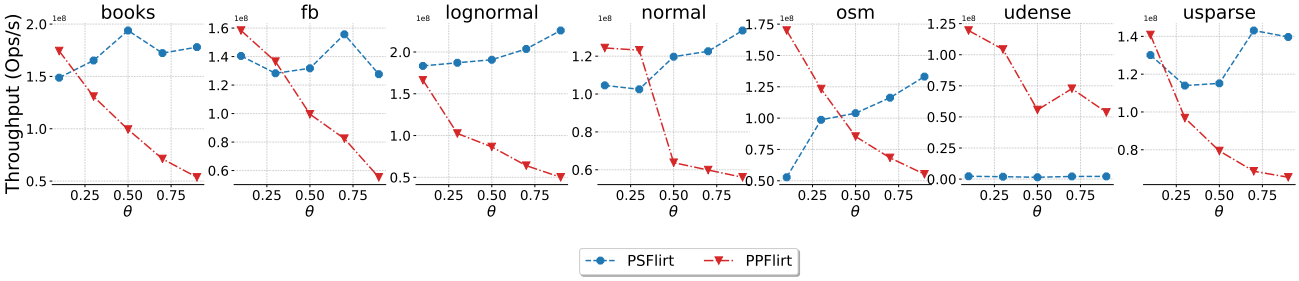

Figure 18: Performance comparison of PPFlirt and PSFlirt for various degrees of skewness θ

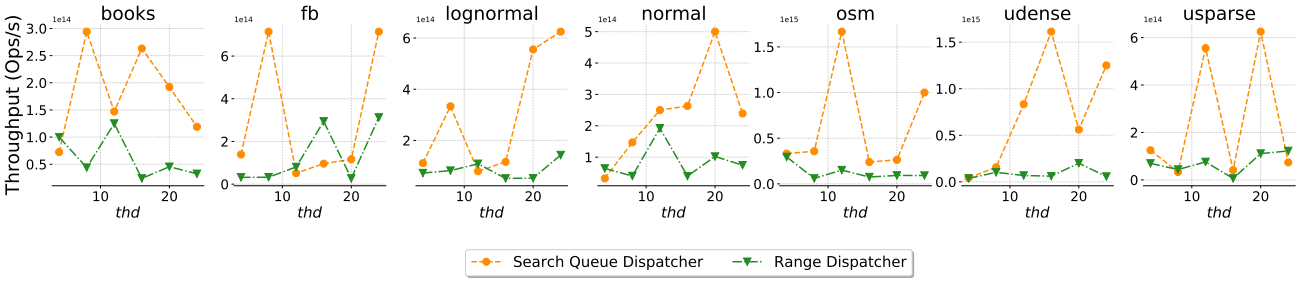

Figure 19: Comparison of different dispatching mechanisms

of searches over the duration. For PPFlirt, each lookup key will be counted once by the thread that processes the key. Figure 17 shows the result.

PPFlirt is more scalable and achieves a much higher search throughput compared to PSFlirt. We expect this result as the search cost for PPFlirt is roughly $\log(N/thd) + \log(Err)$, while the search cost of PSFlirt remains the same as FLIRT at $\log(N) + \log(Err)$. In terms of scalability, PPFlirts achieves an 8.0× increase in search throughput when increasing the thread count from 4 to 24. Combined with the lower execution time in each thread from a smaller index, shown in Figure 13, PPFlirt is able to achieve on average 23.3× speed up over FLIRT. For *uniform dense*, we are able to achieve a super-linear speedup of 32.0×.

PSFlirt, on average, achieves a 2.7× increase in search throughput when increasing the thread count from 4 to 24 and 2.9× speed up over FLIRT. Each thread in PSFlirt does not have the benefit of searching a smaller index and has synchronization overheads (reader-writer locks). When there is one segment in *uniform dense*, all threads access the same segment, which leads to starvation and the performance is shown to be worse than FLIRT. PSFlirt shines in a skewed workload, shown in Figure 18, as it has a better workload balance compared to PPFlirt.

Update performance is shown to suffer compared to FLIRT. For PPFlirt, the overhead comes from the update thread having to search after each update, and the update thread moving between partitions once the current partition is full. For PSFlirt, the overhead is due to synchronization costs. For future work, we will explore other thread configurations to increase update throughput while maintaining high search throughput.

6.10 Query Dispatcher Overhead

We evaluate the performance of the dispatchers for different numbers of threads ranging from 4 to 24. We track the time it takes to complete $50M$ dispatches. For the search queue dispatcher, each partition keeps track of the key range (shown as *min* and *max* in Figure 8). A dispatch operation is complete when the partition that contains the key range processes the key. For the range dispatcher, the dispatching thread keeps track of the key ranges of each partition and forwards the key to the partition that contains the key. The throughput is the number of dispatches over the time taken.

Figure 19 shows the throughput of the dispatchers. The throughput of both dispatchers is in the range of 10^{13} to 10^{14} operations per second, which is approximately 5 to 6 orders of magnitude faster than searching and updating throughput. Therefore, the dispatching mechanism has a negligible overhead in FLIRT operations and will not cause bottlenecks. The search queue dispatcher is shown to outperform the range dispatcher; however, the range dispatcher is shown to be more stable and is consistent with increasing thread count.

7 CONLUSION

In this paper, we present FLIRT, an updatable parameter-free learned index for high-velocity data streams. FLIRT auto-tunes itself to adapt to distribution changes in continuous stream processing. We combine learned indexes with updatable queue structures for fast enqueue, dequeue and search. Specifically, records are stored in linearly predictable segments that allow for the position of records to be estimated by a linear model and require no retraining when updating the index. Our extensive experiments show that FLIRT consistently outperforms traditional indexes, streaming indexes and existing updatable learned indexes, on different datasets, across different workloads and update rates. Two parallel versions of FLIRT, Parallel Partitioned Flirt and Parallel Shared Flirt, are introduced to enhance the search performance for different query workloads. We believe our work is a starting point for introducing learned indexes into streaming-based applications.

REFERENCES

[1] 2013. STX B+ Tree Revisiting Binary Search. https://panthema.net/2013/0504-STX-B+Tree-Binary-vs-Linear-Search/.

[2] Anders Aamand, Piotr Indyk, and Ali Vakilian. 2019. (Learned) Frequency Estimation Algorithms under Zipfian Distribution. *arXiv preprint arXiv:1908.05198* (2019).

[3] 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

[4] Dimitris Bertsimas and Vassilis Digalakis Jr. 2020. Frequency Estimation in Data Streams: Learning the Optimal Hashing Scheme. *arXiv preprint arXiv:2007.09261* (2020).

[5] 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.

[6] 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.

[7] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *SIGMOD*. 1241–1258.

[8] 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 *SIGMOD*. 969–984.

[9] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *PVLDB* 14, 2 (2020), 74–86.

[10] Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. 2015. Ball*-tree: Efficient Spatial Indexing for Constrained Nearest-neighbor Search in Metric Spaces. arXiv:cs.DB/1511.00628

[11] 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 *SIGMOD*. 177–190.

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

[13] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. 1189–1206.

[14] Zdravko Galić. 2016. *Spatio-temporal data streams*. Springer.

[15] Lukasz Golab, Shaveen Garg, and M Tamer Özsu. 2004. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology*. Springer, 712–729.

[16] L. Golab and M.T. Özsu. 2010. *Data Stream Management*. Morgan & Claypool Publishers. https://books.google.co.uk/books?id=IMyogd_LF1cC

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

[18] Ali Hadian, Behzad Ghaffari, Taiyi Wang, and Thomas Heinis. 2021. COAX: Correlation-Aware Indexing on Multidimensional Data with Soft Functional Dependencies. arXiv:cs.DB/2006.16393

[19] Ali Hadian and Thomas Heinis. 2019. Considerations for handling updates in learned index structures. In *AIDM*.

[20] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *EDBT*.

[21] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures. In *AIDB*.

[22] Ali Hadian and Thomas Heinis. 2021. Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction. In *EDBT*.

[23] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB*.

[24] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. 2019. Learning-Based Frequency Estimation Algorithms.. In *ICLR*.

[25] Tanqiu Jiang, Yi Li, Honghao Lin, Yisong Ruan, and David P Woodruff. 2020. Learning-augmented data stream algorithms. *ICLR* (2020).

[26] 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).

[27] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *AIDM*.

[28] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.

[30] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.

[31] 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.

[32] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*.

[33] Anisa Llavesh, Utku Sirin, Robert West, and Anastasia Ailamaki. 2019. Accelerating B+tree Search by Using Simple Machine Learning Techniques. In *AIDB*.

[34] Ahmed R Mahmood, Ahmed M Aly, Tatiana Kuznetsova, Saleh Basalamah, and Walid G Aref. 2018. Disk-based Indexing of Recent Trajectories. *ACM Transactions on Spatial Algorithms and Systems (TSAS)* 4, 3 (2018), 1–27.

[35] Ahmed R Mahmood, Walid G Aref, Ahmed M Aly, and Saleh Basalamah. 2014. Indexing Recent Trajectories of Moving Objects. In *SIGSPATIAL*. 393–396.

[36] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *arXiv:2006.12804* (2020).

[37] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *SIGMOD*. 2789–2792.

[38] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *SIGMOD*. 985–1000.

[39] Vikram Nathan, Jialin Ding, Tim Kraska, and Mohammad Alizadeh. 2020. Cortex: Harnessing Correlations to Boost Query Performance. *arXiv preprint arXiv:2012.06683* (2020).

[40] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The case for learned spatial indexes. In *AIDB*.

[41] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. internal: an essay on machine learning agents for autonomous database management systems. *IEEE bulletin* 42, 2 (2019).

[42] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[43] Naufal Fikri Setiawan, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2020. Function Interpolation for Learned Index Structures. In *ADC*. 68–80.

[44] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-based Stream Join on a Multicore CPU. In *SIGMOD*. 2523–2537.

[45] Manish Singh, Qiang Zhu, and HV Jagadish. 2012. SWST: A Disk Based Index for Sliding Window Spatio-temporal Data. In *ICDE*. 342–353.

[46] Hari Subramoni, Fabrizio Petrini, Virat Agarwal, and Davide Pasetto. 2010. Streaming, low-latency communication in on-line trading systems. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8. https://doi.org/10.1109/IPDPSW.2010.5470717

[47] 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.

[48] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing succinct secondary indexing mechanism by exploiting column correlations. In *SIGMOD*. 1223–1240.

[49] 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.

[50] 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.

[51] 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.