Using Learned Indexes to Improve Time Series Indexing Performance on Embedded Sensor Devices

David Ding, Ivan Carvalho and Ramon Lawrence

University of British Columbia, Kelowna, BC, Canada

Keywords: Learned Indexing, Time Series, Database, Sensor Network, Embedded.

Abstract: Efficiently querying data on embedded sensor and IoT devices is challenging given the very limited memory and CPU resources. With the increasing volumes of collected data, it is critical to process, filter, and manipulate data on the edge devices where it is collected to improve efficiency and reduce network transmissions. Existing embedded index structures do not adapt to the data distribution and characteristics. This paper demonstrates how applying learned indexes that develop space efficient summaries of the data can dramatically improve the query performance and predictability. Learned indexes based on linear approximations can reduce the query I/O by 50 to 90% and improve query throughput by a factor of 2 to 5, while only requiring a few kilobytes of RAM. Experimental results on a variety of time series data sets demonstrate the advantages of learned indexes that considerably improve over the state-of-the-art index algorithms.

1 INTRODUCTION

Embedded systems and IoT devices collect massive amounts of data for use in environmental, industrial, and monitoring applications. This time series data is typically sent over the network to cloud servers for processing. There is increasing focus on processing data on the edge devices where it is collected to reduce network transmissions and power consumption.

There are multiple systems for indexing time series data on cloud servers (Wang et al., 2020; Yang et al., 2019). These systems cannot be adapted to the low-memory and CPU processing present on embedded sensor devices. Specialized embedded indexes such as Microhash (Zeinalipour-Yazti et al., 2005), and sequential binary index for time series (SBITS) (Fazackerley et al., 2021) demonstrate good index performance with minimal memory usage. SBITS allows for both querying by timestamp and data value. Querying by timestamp is done using a linear interpolation search that outperforms binary search used in prior embedded index methods.

Learned indexing is a technique for computing models of the data distribution to efficiently predict record locations. Various techniques have been used including RadixSpline (Kipf et al., 2020) and the Piece-wise Geometric Models (PGM) (Ferragina and Vinciguerra, 2020). These indexes outperform traditional tree-based indexes for server applications. There has been no prior work on adapting learned indexes for embedded indexing.

The contribution of this work is the adaptation of learned indexes for use with time series data collected and processed by sensor devices. Learned indexing techniques using RadixSpline and Piece-wise Geometric Models are implemented and evaluated on actual sensor devices with real-world data sets. Experimental results show a dramatic improvement of time series query performance by a factor of two to five times compared to binary search and linear interpolation used by SBITS. With a small amount of index memory, the query performance is significantly improved demonstrating that learned indexes are valuable for the sensor domain.

2 BACKGROUND

Indexing on flash memory has unique characteristics, and many index algorithms have been developed to optimize performance (Fevgas et al., 2020). Indexing on sensor devices is typically on flash memory and requires additional optimizations beyond server-based indexing. Data structures must adapt to the unique performance characteristics of flash memory such as different read and write times and the erase-before-write constraint (i.e. no overwriting or in-place writes). Common optimizations minimize writes and use sequential rather than random writes.

A key consideration for sensor indexing is the limited resources. Devices have between 4 KB and 32 KB of SRAM memory, a processor running between

16 and 128 MHz, and flash storage consisting of a SD card or raw flash memory chips. Index structures must minimize SRAM usage to be practical.

Data processing systems for embedded devices such as Antelope (Tsiftes and Dunkels, 2011) and LittleD (Douglas and Lawrence, 2014) allow on-device data processing. Improved query performance is achievable by indexing the time series data. Antelope (Tsiftes and Dunkels, 2011) supports an inline index sorted by timestamp. Record lookup by timestamp is performed using binary search. MicroHash (Zeinalipour-Yazti et al., 2005) also stores data as a sorted time series data file and utilizes binary search for timestamp queries. MicroHash uses range partitioning to also index data values to support queries by data value. PBFilter (Yin and Pucheral, 2012) uses Bloom filters to summarize data for indexing. By sequentially writing the data and index files, PBFilter outperforms tree and hashing techniques.

SBITS (Fazackerley et al., 2021) is a sequential indexing structure supporting timestamp and data value based queries. Timestamp queries are efficiently done using linear interpolation. The location of a record is predicted based on its timestamp and a linear approximation of the data records stored. Given that sensor data sets are often collected at consistent intervals, location predication using linear interpolation is highly effective and outperforms binary search. Since data pages are ordered by timestamp, any timestamp can be retrieved in $O(1)$ reads by calculating an offset using the start timestamp, the query timestamp, and the rate at which records are written to data file (based on sampling rate and number of records per block). The linear interpolation algorithm can be extended to support multiple different sensor sampling rates and time periods by storing a linear approximation for each period, however that is not in the current implementation. SBITS also supports efficient querying by data value by using a user-customizable bitmap index. Prior experimentation (Ould-Khessal et al., 2022) demonstrated that SBITS outperforms MicroHash and other embedded indexing techniques.

Given the relative consistency of sensor timeseries data, linear interpolation for location prediction is effective. However, there are cases in real-world data sets where the data is not as regular and cannot be easily approximated by a single linear function. Events such as changing the sampling frequency, power failures, or sensor outages may cause the data set to be less than regular and index performance to degrade. It is valuable to consider improved location predication techniques such as offered by learned indexes.

Learned indexes leverage machine learning models to outperform traditional indexes and were ini-

tially used for static, in-memory datasets (Kraska et al., 2018). Early work on learned indexes focused on modeling the empirical Cumulative Distribution Function (eCDF). Examples of such indexes include the RMI (Kraska et al., 2018), RadixSpline (Kipf et al., 2020), PGM Index (Ferragina and Vinciguerra, 2020), and PLEX (Stoian et al., 2021).

The core idea of those learned indexes is the connection between the eCDF and an element's position in a sorted array. Let $P(A \leq x)$ be the proportion of elements smaller than a key x. Then, the position of the key x in a sorted array with N elements is $pos = \lfloor N \cdot P(A \leq x) \rfloor$. Obtaining an exact formula for $P(A \leq x)$ is challenging, hence the indexes opt to train different kinds of machine learning models to estimate the eCDF.

eCDF based learned indexes are highly effective for static, in-memory use cases. Marcus et al. benchmarked the RMI, the PGM, and the RadixSpline against state-of-the-art traditional indexes on real-world datasets (Marcus et al., 2020). Their experimental results show that the learned indexes always outperform traditional indexes on look-up time and size, losing just on build time.

Although learned indexes are highly effective on read-only workloads, adapting the indexes to handle updates is a challenge. This is particularly relevant for the sensor use case, as sensors are continually collecting and indexing data. The index must handle a continuous stream of appended data.

Learned indexes such as ALEX (Ding et al., 2020) and LIPP (Wu et al., 2021) support updates. These indexes are tree-based, and use the models to search the tree. Instead of only approximating the eCDF, tree-based learned indexes also focus on using the models to partition the data evenly and achieve a small tree height. The consequences of a smaller tree height are faster lookups and a smaller memory footprint.

Despite the progress of updatable learned indexes, they do not always beat traditional indexes in mixed read-and-write scenarios. Wongkham et al. tested ALEX, LIPP, and an updatable version of the PGM in multiple scenarios in their benchmark (Wongkham et al., 2022). Among their conclusions is that space efficiency is not a guarantee of updatable learned indexes. The gains in memory consumption are lowered for ALEX and PGM under updates, and the LIPP consumes more memory than traditional indexes. This could potentially rule out applications of learned indexes to sensors, as they are constantly updated and have very limited memory resources.

3 ADAPTING LEARNED INDEXES

Applying learned indexing structures for sensor time series data requires supporting the continual increase of the data size as new values are collected. Therefore, we need to adapt learned indexes to support append operations while keeping memory consumption low. Further, learned indexes only apply to indexing the sensor data by timestamp as records are stored in a sorted file by timestamp as they are collected. Learned indexes do not directly apply to indexing the data (sensor values) in the record, which are not stored in sorted order.

We focus on adapting bottom-up, eCDF based learned indexes. Indexes that are built bottom-up such as the RadixSpline do not require all the data in advance, in contrast to top-down modeling approaches such as RMIs that require all data in advance and cannot be easily deployed for the time series use case. We adapt and deploy two structures for indexing: the RadixSpline and the PGM Index.

The indexing approach is based on SBITS (Fazackerley et al., 2021) and proceeds as follows:

- Each data record containing collected sensor values is stored in a buffered page in memory until the page is full.

- A full data page is written to the sorted data file in sequential order.

- After the page is written, an index record is created storing the timestamp of the smallest record on the data page and the data page index.

- The index algorithm must maintain its index structure in memory with a bounded size (often less than 1 KB) and may periodically flush index pages to storage for persistence.

Querying by timestamp is performed by using the index. Querying by data value is unchanged from the previous SBITS implementation as learned indexes do not apply to the unsorted data values.

3.1 Piece-Wise Geometric Models

The Piece-wise Geometric Model index (PGM) is a learned index that approximates the CDF via piece-wise linear approximations (PLA) (Ferragina and Vinciguerra, 2020). At the heart of the PGM lies a hyperparemeter ε that controls the error bounds for the linear approximations.

The original PGM index is built from the bottom-up recursively. At the first level, the PGM builds a PLA over the set of points $\{(x_i, i)\}_{i=0...n}$ using the optimal algorithm proposed by (O'Rourke, 1981) and

rediscovered by (Elmeleegy et al., 2009). For subsequent levels, the PGM applies the same linear approximation strategy using the keys from the previous level $\{(k_j, j)\}_{j=0...s}$ as points for the PLA. The recursion halts when there is a level with exactly one line.

We adapt the PGM by creating an append method described in Algorithm 1. Notice that the method does not know the number of levels for the PGM in advance and starts by appending a key to the linear approximation at the bottom level and propagating the updates to the upper levels if necessary. Another modification of our implementation is the use of the Swing Filter algorithm also proposed by (Elmeleegy et al., 2009). The Swing Filter does not yield an optimal number of lines, but requires only $O(1)$ time and memory to process a point. This is a benefit in the sensor use case, as the optimal Slide Filter requires $O(h)$ memory in its worst-case where h is the number of points in the convex hull, which might not fit in-memory. Even though we adapted the PGM to support appends, its procedure for finding a key remains unchanged from the original PGM (see Algorithm 2).

Algorithm 1: AppendPGMAdd(Pgm, x).

$L \leftarrow$ pgm.countLevels();
$K \leftarrow x$;
for $i \leftarrow 0$ *to* $L - 1$ **do**
    $c \leftarrow pgm.levels[i].countPoints()$;
    pgm.levels[i].add(K); // add
        implements the Swing Filter
    if $pgm.levels[i].countPoints() > c$ **then**
        $K \leftarrow$ pgm.levels[i].getLastKey();
    else
        return;

/* If the algorithm reached this
   step we need to create a new
   level   */
pgm.levels[L] = newPGMLevel();
firstK $\leftarrow$ pgm.levels[L-1].getFirstKey();
lastK $\leftarrow$ pgm.levels[L-1].getLastKey();
pgm.levels[L].add(firstK);
pgm.levels[L].add(lastK);
return;

3.2 RadixSpline

The RadixSpline is a learned index that approximates the CDF via a linear spline and a radix table storing spline points (Kipf et al., 2020). The two hyperparameters that shape the RadixSpline are ε, the error bound for the spline approximation, and r, the size of the prefix of the radix table entries.

The RadixSpline builds an error-bounded linear

Algorithm 2: QueryAppendPGM(Pgm, *x*).

> **if** *x < pgm.levels[0].getFirstKey()* **then**
>   **return** *NotFound*;
> $L \leftarrow$ pgm.countLevels();
> m$\leftarrow$ 0; // index of the model at the
>   i-th level
> **for** $i \leftarrow L - 1$ *to* 1 **do**
> $a \leftarrow$ pgm.levels[i].getSlope(m);
> $b \leftarrow$ pgm.levels[i].getIntercept(m);
>   pos $\leftarrow \lfloor a \cdot x + b \rfloor$;
>   lo $\leftarrow \max\{0, pos - \varepsilon - 1\}$;
>   hi $\leftarrow \min\{pgm.levels[i].countKeys() -$
> $1, pos + \varepsilon + 1\}$;
>   m $\leftarrow$ smallest value of *j* such that $x \geq$
>   pgm.levels[i-1].getKey(j) and
> $lo \leq j \leq hi$;
> $a \leftarrow$ pgm.levels[0].getSlope(m);
> $b \leftarrow$ pgm.levels[0].getIntercept(m);
> pos $\leftarrow \lfloor a \cdot x + b \rfloor$;
> lo $\leftarrow \max\{0, pos - \varepsilon - 1\}$;
> hi $\leftarrow \min\{pos + \varepsilon +$
> $1, pgm.levels[0].countKeys() - 1, \}$;
> /* Search is an implementation of
>   binary search or linear search
>   over the original array
>   containing the keys   */
> **return** *Search(x, lo, hi)*;

spline over the empirical CDF using the greedy algorithm proposed by (Neumann and Michel, 2008). After the spline is built, the prefixes of the spline points are inserted in a radix table. The radix table is a flat array of size 2^r where each entry in the table maps to a range in the spline.

Querying for a point in a RadixSpline is done in three steps. The first step is to look for the prefix of the key being searched in the radix table, and find the corresponding spline point. Given the spline point, calculate a narrow range of size 2ε where the key could be using linear interpolation. The last step is to find the key in the range using linear or binary search, which is a quick operation because 2ε is constant.

We adapted the RadixSpline to support appends by implementing a streaming version of the GreedySpline proposed in (Neumann and Michel, 2008), as described in Algorithm 3. After appending a point, we check if we can adjust the last spline segment to correctly approximate the point position within ε. If that is not the case, we create a new spline segment covering the new point and propagate the new spline point to the radix table (see Algorithm 4).

Algorithm 3: RadixSplineAdd(Rs, *x*).

> $c \leftarrow$ rs.spline.countPoints();
> rs.spline.add(x); // Add implements
>   GreedySpline
> **if** *rs.spline.countPoints() > c* **then**
> $K \leftarrow$ rs.spline.getLastKey();
>   RadixTableInsert(rs, K, c); // see
>   Algorithm 4
>   **return**;
> **return**;

Algorithm 4: RadixTableInsert(Rs, K, pos).

> $r \leftarrow$ rs.r; // radix prefix size
> $K \leftarrow$ K - rs.minKey;
> nB $\leftarrow$ mostSignificantBit(bitShiftRight(K,
>   rs.shiftSize)); // number of bits
>   required to fit on the table
> $\Delta \leftarrow$ max(nB - r, 0); // difference
>   between the available and
>   required number of bits
> **if** $\Delta > 0$ **then**
>   /* New key triggers table
>   rebuild with new prefix size
>   in order to fit; we merge the
>   old radix entries considering
>   the new bit shift   */
>   rs.shiftSize $\leftarrow$ rs.shiftSize + Δ;
>   newTable $\leftarrow$ allocateTable(2^r);
>   **for** $i \leftarrow 0$ *to* 2^r **do**
>   j $\leftarrow$ bitShiftRight(i, Δ);
>   newTable[j] $\leftarrow$ min(rs.table[i],
>   newTable[j]);
>   **for** $i \leftarrow 2^{r-\Delta}$ *to* 2^r **do**
>   newTable[i] $\leftarrow$ INT_MAX;
>   rs.table $\leftarrow$ newTable;
> // Update entry of the radix table
>   with the new pos
> $T \leftarrow$ bitShiftRight(K, rs.prefixSize);
> rs.table[T] $\leftarrow$ min(rs.table[T], pos);
> **return**;

4 EXPERIMENTAL RESULTS

The experiments evaluated sensor time series indexing for multiple real-world data sets. The sensor hardware platform has a 32-bit Microchip ARM® Cortex® M0+ based SAMD21 processor with clock speed of 48 MHz, 256 KB of flash program memory and 32 KB of SRAM. The hardware board has several different memory types including a SD card

and serial NOR DataFlash[1] which supports in-place page level erase-before-write. This platform is representative of embedded devices with commonly used 32-bit ARM processors. The serial NOR DataFlash was used to test performance on raw memory without a flash translation layer (FTL). Testing on raw memory insures no overhead with FTL maintenance operations and allows for the highest read performance. Platform performance characteristics are in Table 1.

The experiments benchmark four indexes:

- **Binary Search**: A binary search over the sorted data. This is the baseline for the benchmark and requires no additional memory.

- **SBITS**: SBITS using linear interpolation requires 8 bytes to maintain linear approximation. SBITS was optimized to default to binary search if its predictions were off significantly.

- **PGM**: A modified version of the PGM Index with support for appends and error-bound of $\varepsilon = 1$.

- **RadixSpline**: A modified version of the RadixSpline with support for appends, error-bound set to $\varepsilon = 1$ and radix prefix size of $r = 0$ that offered the best performance with smallest memory usage.

The experiments measure four core metrics: the query throughput, the number of I/Os per query, memory consumption, and insertion throughput. These metrics are relevant to common use cases of sensors such as querying by timestamp and ingesting new data. We report the average of the metrics based on three separate runs. The real-world data sets evaluated are in Table 2. The data sets cover a variety of sensor use cases including environmental monitoring, smart watches, GPS phone data, and chemical concentration monitoring. The environmental data sets **sea** and **uwa** were originally from (Zeinalipour-Yazti et al., 2005) and have been used in several further experimental comparisons (Fazackerley et al., 2021; Ould-Khessal et al., 2022). We also present the eCDF for the datasets in Figure 1.

4.1 Query Performance

One common use case of processing data in edge devices is to query timestamps. We measure the query throughput for searching timestamps. After the sensor data was inserted, 10,000 random timestamp values were queried in the timestamp range. Data was collected on the time to execute all queries and the number of I/O operations performed.

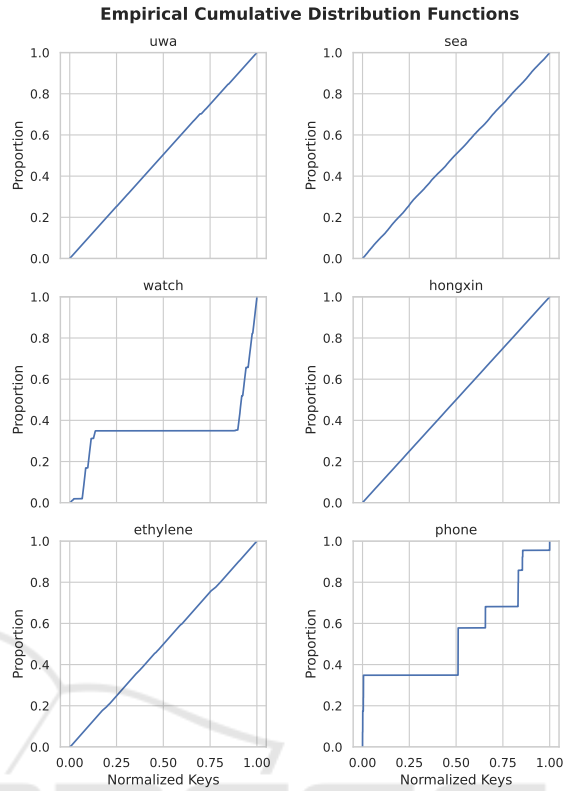[1]https://www.dialog-semiconductor.com/products/memory/dataflash-spi-memory

Figure 1: eCDFs for the experimental datasets. The PGM and the RadixSpline use a collection of linear models to approximate those with bounded-error of ε.
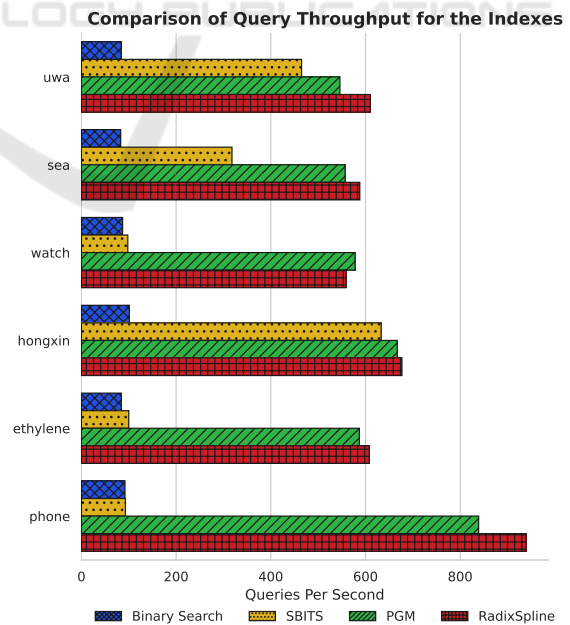

Figure 2: Average query throughput among the indexes for each dataset in the benchmark. Higher rates indicate better results. The RadixSpline and PGM consistently outperform SBITS and binary search.

Table 1: Hardware Performance Characteristics.

	Reads (KB/s)		Writes (KB/s)		
	Seq	Random	Seq	Random	Write-Read Ratio
M0+ SAMD21 (DataFlash)	475	475	38	38	12.5

Table 2: Experimental Data Sets.

Name	Points	Points Used	Sensor Data	Source
sea	100,000	100,000	temp, humidity, wind, pressure	SeaTac Airport
uwa	500,000	500,000	temp, humidity, wind, pressure	ATG rooftop, U. of Wash.
hongxin	35,064	35,000	PM2.5, PM10, temp	(Zhang et al., 2017)
ethylene	4,085,589	100,000	ethylene concentration	(Fonollosa et al., 2015)
phone	18,354	18,000	smartphone X/Y/Z magnetic field	(Barsocchi et al., 2016)
watch	2,865,713	100,000	smartwatch X/Y/Z gyroscope	(Stisen et al., 2015)

The query throughput results are in Figure 2. Applying learned indexes to sensor time series is very effective. The PGM and the RadixSpline always outperform SBITS with significant improvements on highly variable datasets.

For the **hongxin** and **uwa** datasets, the gains are more modest ranging from 1.05x-1.20x for the PGM and 1.06x-1.31x for the RadixSpline. These datasets are highly linear such that an interpolation search achieves its best-case scenario. Since the PGM and RadixSpline use PLAs to model the dataset, they also perform well just like the interpolation search.

However, datasets such as **watch**, **ethylene**, and **phone** prove to be more challenging for SBITS. Its performance is affected and becomes closer to the throughput of the binary search baseline. Learned indexes, on the other hand, extend their lead and stay resilient to the change in data distribution. The performance gains range from 1.8x-8.9x for the PGM and 1.8x-10x for the RadixSpline. This indicates that the learned indexes have more flexible models that describe the data distribution better.

The query throughput difference between RadixSpline and PGM is within 10% for all data sets. Since both approaches use linear approximations, the difference relates to how the linear approximations are themselves indexed. For these experiments, the radix table for RadixSpline was allocated no space, and only a binary search was used on the spline points. This is effective as there are very few points. Performance testing with using a radix table of size $r = 4$ and $r = 8$ demonstrated no query performance benefit while consuming precious RAM. This makes sense as the radix table is only saving a few comparisons when searching the spline points in memory, and the search time is dominated by the I/Os performed to the flash memory. PGM produces a multi-level index, which takes some more space and a little longer to query. Overall, both approaches are effective and
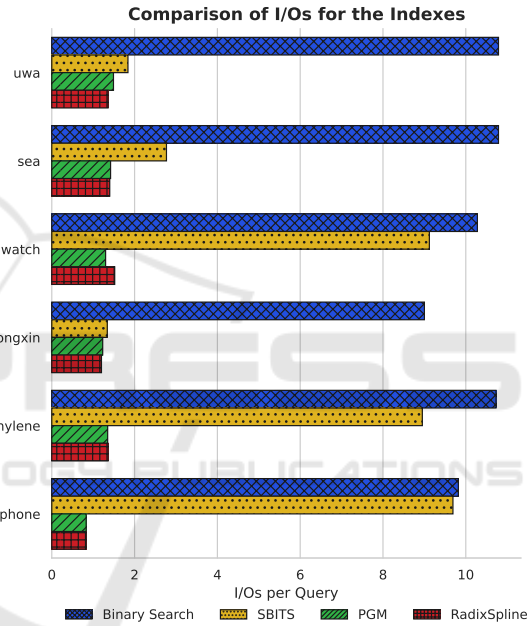

Figure 3: Average number of I/Os per query among the indexes for each dataset in the benchmark. Lower rates are better. Learned indexes significantly reduce the number of required I/Os.

greatly improve on binary search or single linear interpolation. The trade-off between the query performance and memory space is discussed further in Section 4.2.

Another relevant metric for querying is the number of I/Os per query. The number of I/Os performed dominates the query response time and is especially important in embedded systems were predictable real-time performance is desirable. Figure 3 displays the average number of I/Os performed per timestamp query. Binary search performs $O(\log N)$ I/Os and has significantly more I/Os than the other approaches. SBITS' single linear interpolation is effective in many cases, however data sets like phone are

poorly approximated by one linear approximation and the algorithm frequently defaults to binary search due to poor prediction accuracy. Both PGM and RadixSpline have guaranteed error bounds in their construction. With $\varepsilon = 1$, at most 2 I/O are performed for any lookup with the average often around 1.3 to 1.5. This predictable performance is a major benefit for using these learned indexes.

4.2 Memory Space Efficiency

It is critical for learned indexes to have a small memory footprint in order to be useful for embedded systems. Many traditional techniques cannot be applied to embedded systems because of the RAM constraints (Fazackerley et al., 2021), which prompts adaptations that trade memory and performance. The memory results are in Table 3.

Table 3: Memory consumption comparison among the learned indexes for each dataset in the benchmark.

Memory Consumption (in KB)		
	PGM	**RadixSpline**
uwa	0.04	0.09
sea	1.88	0.96
watch	7.80	2.43
hongxin	0.10	0.07
ethylene	0.40	0.12
phone	1.09	0.25

All indexes fit in memory, with the maximum amount of memory used being 7.80 KB. The results indicate that the RadixSpline consumes less memory than the PGM. This is to be expected, as both the spline and the bottom level of the PGM contain a very similar set of linear approximations. The key difference between the two is their approach to finding the linear approximation to use. The RadixSpline uses a small radix table to index spline points. The PGM uses a recursive approach building additional PLAs. Since the size of the table from the RadixSpline is parametrized by r, it is possible to sacrifice a little query performance to achieve less memory usage.

For our choice of $r = 0$, the difference in memory consumption is significant, and the difference in query performance is negligible as performance is dominated by the number of I/Os not comparisons in memory. The PGM uses two to four times the amount of memory as the RadixSpline for five out of the six datasets, with the exception being the easiest dataset **uwa**.

By modifying the error bound (ε), both approaches can reduce their memory footprint at the sacrifice of more query I/Os and lower query through-

put. Table 4 shows statistics on the index size in bytes, I/Os performed per query, and query throughput in queries/second for the **sea** data set for multiple different values of ε. There is a quite significant index size reduction for increasing ε to 2 or 3. Even though the I/Os per query increases, it is always bounded by ε. This allows designers to determine the exact performance trade-offs in terms of space and query I/Os in a predictable fashion.

4.3 Insertion Performance

Adding indexes benefits query performance, but they can also impact the insertion time. It is critical for sensors to keep ingesting data; hence the need to monitor the insertion throughput to ensure they match the sensors required sampling rate. For insertion performance, the N records used for each data set were inserted at the maximum possible rate of the hardware. The insertion performance is dominated by the I/Os for writing the data pages to storage, but the index construction time may have some overhead. The insertion rates in Figure 4 show the maximum rates possible on the hardware for each data set and index.

The baseline for insertion performance is the binary search case which consists of a simple data record append and no indexing overhead. The average throughput was 1967 inserts per second. Since the incoming timestamps are strictly increasing and because binary search does not store any additional information to help on the search, this represents the upper bound for insertion performance.

SBITS and RadixSpline were the indexes that had the highest insertion throughputs at 1966 and 1965 inserts per second, respectively. They almost match the scenario with no index at all. The results are consistent with earlier experiments for SBITS (Ould-Khessal et al., 2022), showing that the number of I/Os to keep the index up to date is minimal. Perhaps the most surprising result is for the RadixSpline, because it needs to calculate spline points and update the radix table. Our benchmark shows that the overhead for these operations are small.

The PGM supports 1951 inserts per second, which represents a less than 1% overhead. The PGM has a slightly lower insert rate because some insertions trigger changes on multiple levels of the PGM, while RadixSpline triggers at most one change in the spline and radix table. The PGM insertion throughput remains competitive beating traditional embedded indexes such as B-Trees (Ould-Khessal et al., 2022).

Table 4: Index Size in bytes, I/Os per Timestamp Query, and Query Throughput (queries/sec.) for Different Error Bounds ε.

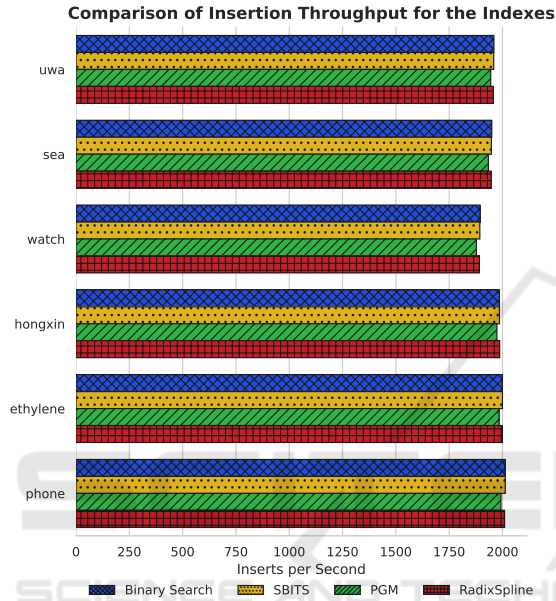| Test	RadixSpline			PGM		
	Index Size	Query I/O	Throughput	Index Size	Query I/O	Throughput
SEA ε = 1	932	1.40	588	1920	1.42	557
SEA ε = 2	436	1.85	461	856	1.91	440
SEA ε = 3	260	2.23	386	664	2.26	376
SEA ε = 5	212	3.12	281	496	3.41	256
SEA ε = 10	132	5.37	166	232	4.94	179

Figure 4: Average insertion throughput among the indexes for each dataset in the benchmark. Higher rates indicate better results. The insertion throughout is slightly lower for the PGM, but overall learned indexes remain competitive.

4.4 Results Discussion

Overall, the experimental results demonstrate that there are significant advantages to using learned indexes adapted for embedded time series data. The most significant advantage is the predictable and bounded timestamp query performance. By specifying a given error bound (ε), the maximum number of I/Os per query is $2ε + 2$. The query performance is significantly higher than SBITS linear interpolation search or binary search. The overhead of the index in terms of insertion throughput is minimal. To handle the limited memory, the index size can be reduced by increasing the error bound. In the experiments tested, the index size was usually less than a few KB. The index algorithms support real-time sensor data collection of almost 2000 records/second on the experimental hardware, which is significantly above collection rates for most applications.

5 CONCLUSIONS

This work adapted and applied learned indexes to sensor time series data. Experiments indicate that the RadixSpline and PGM are suitable to tackle the challenge of indexing the amount of data embedded systems collect under the unique constraints that those systems have. This work is a bridge between embedded system indexing and the original applications of learned indexing targeting large in-memory databases. With the correct adaptations, it is possible to reuse these structures conceived to query large static datasets and apply them in embedded systems with tiny memory and frequent data appends.

Future work will extend learned indexes for sensors to automatically handle hyperparameters and require less tuning. We envision learned indexes that automatically adapt their parameters to fit in the limited resources in embedded systems, specifically adapting the error bound to ensure the index fits in the memory available.

REFERENCES

Barsocchi, P., Crivello, A., Rosa, D. L., and Palumbo, F. (2016). A multisource and multivariate dataset for indoor localization methods based on WLAN and geomagnetic field fingerprinting. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE.

Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., and Kraska, T. (2020). ALEX: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984. ACM.

Douglas, G. and Lawrence, R. (2014). LittleD: a SQL database for sensor nodes and embedded applications. In *Symposium on Applied Computing*, pages 827–832.

Elmeleegy, H., Elmagarmid, A. K., Cecchet, E., Aref, W. G., and Zwaenepoel, W. (2009). Online piece-wise linear approximation of numerical streams with precision guarantees. *Proc. VLDB Endow.*, 2(1):145–156.

Fazackerley, S., Ould-Khessal, N., and Lawrence, R.

(2021). Efficient flash indexing for time series data on memory-constrained embedded sensor devices. In *Proceedings of the 10th International Conference on Sensor Networks, SENSORNETS 2021*, pages 92–99. SCITEPRESS.

Ferragina, P. and Vinciguerra, G. (2020). The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175.

Fevgas, A., Akritidis, L., Bozanis, P., and Manolopoulos, Y. (2020). Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *VLDB J.*, 29(1):273–311.

Fonollosa, J., Sheik, S., Huerta, R., and Marco, S. (2015). Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629.

Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., and Neumann, T. (2020). RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. (2018). The case for learned index structures. In *Proceedings of the 2018 SIGMOD International Conference on Management of Data*, pages 489–504.

Marcus, R., Kipf, A., van Renen, A., Stoian, M., Misra, S., Kemper, A., Neumann, T., and Kraska, T. (2020). Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13.

Neumann, T. and Michel, S. (2008). Smooth interpolating histograms with error guarantees. In *Lecture Notes in Computer Science*, pages 126–138. Springer Berlin Heidelberg.

O'Rourke, J. (1981). An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578.

Ould-Khessal, N., Fazackerley, S., and Lawrence, R. (2022). Performance Evaluation of Embedded Time Series Indexes Using Bitmaps, Partitioning, and Trees. In *Invited and revised papers of SENSORNETS 2021*, volume 1674 of *Sensor Networks*, pages 125–151. Springer.

Stisen, A., Blunck, H., Bhattacharya, S., Prentow, T. S., Kjærgaard, M. B., Dey, A., Sonne, T., and Jensen, M. M. (2015). Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, page 127–140, New York, USA. ACM.

Stoian, M., Kipf, A., Marcus, R., and Kraska, T. (2021). Towards practical learned indexing. *https://arxiv.org/abs/2108.05117*.

Tsiftes, N. and Dunkels, A. (2011). A database in every sensor. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, page 316–332, New York, NY, USA. ACM.

Wang, C., Huang, X., Qiao, J., Jiang, T., Rui, L., Zhang, J., Kang, R., Feinauer, J., Mcgrail, K., Wang, P., Luo, D., Yuan, J., Wang, J., and Sun, J. (2020). Apache IoTDB: Time-series database for Internet of Things. *Proc. VLDB Endow.*, 13(12):2901–2904.

Wongkham, C., Lu, B., Liu, C., Zhong, Z., Lo, E., and Wang, T. (2022). Are updatable learned indexes ready? *Proceedings of the VLDB Endowment*, 15(11):3004–3017.

Wu, J., Zhang, Y., Chen, S., Wang, J., Chen, Y., and Xing, C. (2021). Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288.

Yang, Y., Cao, Q., and Jiang, H. (2019). EdgeDB: An Efficient Time-Series Database for Edge Computing. *IEEE Access*, 7:142295–142307.

Yin, S. and Pucheral, P. (2012). PBFilter: A flash-based indexing scheme for embedded systems. *Information Systems*, 37(7):634 – 653.

Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. (2005). MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*, pages 31–43. USENIX Association.

Zhang, S., Guo, B., Dong, A., He, J., Xu, Z., and Chen, S. X. (2017). Cautionary tales on air-quality improvement in Beijing. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2205):20170457.