Applying Learned Indexing on Embedded Devices for Time Series Data

by

Yiming Ding

B.Sc., The University of British Columbia, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The College of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

July, 2023

© Yiming Ding, 2023

The following individuals certify that they have read, and recommend to the College of Graduate Studies for acceptance, a thesis/dissertation entitled:

Applying Learned Indexing on Embedded Devices for Time Series Data

submitted by YIMING DING in partial fulfilment of the requirements of the degree of Master of Science.

Dr. Ramon Lawrence, Irving K. Barber Faculty of Science
Supervisor

Dr. Khalad Hasan, Irving K. Barber Faculty of Science
Supervisory Committee Member

Dr. Yong Gao, Irving K. Barber Faculty of Science
Supervisory Committee Member

Dr. Barb Marcolin, Faculty of Management
University Examiner

Abstract

To meet the demand for increasingly accurate sensor monitoring and forecasting, time series datasets have grown to take larger, more detailed samples with more frequent sampling rates. As a result, the size of time series datasets has grown larger and now require more efficient indexing methods to manage properly. Time series databases have unique traits that allow for efficient indexing structures to be used. The timestamps are always increasing in an append-only fashion, a characteristic which can be exploited to create more efficient indexing structures. In this research, two different indexing algorithms for time series databases are evaluated. The spline index model uses existing points of the time series data to form a series of linear approximations. The other index model examined is the piece-wise geometric model (PGM), which forms fully independent lines that approximate the underlying time series data. Experimental results show both the Spline and PGM learned indexes outperform conventional indexes for time series data. Performance metrics for binary search and simpler single line approximations are also included for comparison.

Lay Summary

Wireless sensor networks monitor and collect data in areas too large to be covered by a single sensor, or in scenarios where it is impractical to run cables to every individual sensor device. Sensor network nodes are comprised of battery-powered embedded devices with flash storage that communicate using a wireless radio. Data gathered by sensor nodes can be transmitted directly to a base station where it is stored and processed, however, lower energy consumption (and longer battery life) can be achieved by processing the data locally. Local data processing requires an efficient data indexing structure optimized for flash memory with the limited CPU and memory resources available to an embedded device. In this work we modify two learned indexing structures (Spline and PGM) to support the continuous appending functionality required for continuously generated sensor data. Benchmark results on ordered data sets show performance advantages compared with conventional indexes.

Preface

This work has been published in the following publication:

David Ding, Ivan Carvalho, and Ramon Lawrence. Using Learned Indexes to Improve Time Series Indexing Performance on Embedded Sensor Devices. In Proceedings of the 12th International Conference on Sensor Networks, February 2023, ISSN 2184-4380, ISBN: 978-989-758-635-4, DOI: 10.5220/0011692900003399, pages 23-31.

Some components of this thesis are from this publication.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I deeply appreciate the support and guidance that I have received from Dr. Ramon Lawrence over the course of my time at UBC, his enthusiasm for teaching and genuine care for his students is obvious and continues to be an inspiration.

Many thanks to Ivan Carvalho, whose contributions to the PGM implementation and published paper are greatly appreciated and were instrumental in helping to keep things on track.

Dedication

This work is dedicated to mom and dad, whose love and continued support of my education made it possible for me to go as far as I've come.

Chapter 1

Introduction

A time series is a set of data indexed using temporal information. This type of data set is generally sorted by their temporal indexes and are commonly used to track metrics over a period of time. A typical time series data point is comprised of a data segment, coupled with a temporal index. The data segment can contain arbitrary information and can contain multiple fields. The temporal index must contain some descriptor of the time associated with the data segment. Data organized in this way can be used to monitor changes, extrapolate trends, and find patterns that emerge when comparing data points against themselves at different points in time.

Time series databases are software solutions designed to handle time series data and provide the means to efficiently insert new data points and to query for existing data within a time series dataset. Time series databases are commonly applied in fields such as meteorology, finance, and economics where time-dependent data such as stock prices, climate information, and macroeconomic data can be stored and analyzed. These are fields in which forecasting is a major point of interest, and in which significant value can be extracted from correct predictions of future trends.

They are also used in a variety of other applications with a focus on monitoring. Examples include manufacturing, power grids, and retail. In these scenarios, reacting to changing conditions in real time is a major interest, and time series databases are able to send out notifications to the end users if production falls too far behind schedule, power demands spike unexpectedly, or if product inventory falls to a critical level. Generally, organizations use time series databases to gain insights into their data, predict about future trends, and optimize ongoing operations.

A notable use-case example of time series data logging is their applications in agriculture. Deployments of embedded sensor devices could directly reduce operating costs and increase profits from crop yields by continuously sampling the environment to generate time series data. Soil moisture sensors can enable the use of targeted irrigation, reducing costs by watering crops only in parts where it is necessary. Temperature, humidity, and precipitation data can inform personnel of abnormal weather conditions so they have a

chance to take preventative measures for preventing crop loss.

Time series data is typically collected with a consistent sampling interval, the result of which are timestamps that are equally spaced apart. These data points are then stored on disk for later retrieval. By applying a mathematical model on top of the timestamps, it is possible to estimate the approximate index of a specific data slice. Indexes can then be used to generate the disk location of a specific data slice, which reduces the number of disk access and comparison operations needed to complete a database search.

When time series are sampled regularly with zero missed sampling points, a simple linear model would be sufficient to accurately predict the position of any data point. However, time series datasets in practice often have missed samples, or are only sparsely populated on demand. For example, meteorological stations may opt to skip certain reporting intervals due to maintenance operations or equipment failure. Additionally, in applications where energy efficiency is critical (such as with battery powered devices), a design decision may be implemented to only record new data samples when noteworthy observations are made. With these sparsely populated time series, a more sophisticated indexing method is necessary.

Machine learning models have seen significant research in recent years regarding their applicability in the enterprise database area [KBC+17]. These models aim to improve upon the performance offered by traditional indexing methods by fitting the patterns in the underlying distribution of the data. Piecewise linear functions are a type of learned indexes that have been applied in the field of machine learning and big data as an efficient method of approximating a Cumulative Distribution Function (CDF). The memory efficiency of these models makes it possible to index massive datasets where a conventional index may not fit in memory [ALAB+20], but it could also lend itself well in situations where limited hardware resources are available.

This work adapts learned indexing techniques, specifically the Spline and PGM piecewise linear functions, originally designed for server applications for indexing time series data on embedded devices. A key contribution is supporting incremental appends as a sensor collects time series data over time. The performance of these index structures is benchmarked against other indexing methods using metrics such as memory consumption, IO count, and throughput. We demonstrate the advantages of applying a learned index for time series data supported by our experimental benchmarking results on real-world datasets.

The contributions of this thesis are:

– A C implementation of the Spline and PGM learned indexes for limited

memory embedded systems

- The addition of append support for the Spline and PGM indexes
- Performance evaluations of the Spline and PGM learned indexes against the SBITS conventional index for multiple real-world sensor data sets

The organization of this thesis is as follows. Chapter 2 provides background on database systems and indexing with specific details on unique challenges and solutions for indexing on embedded devices and sensor networks. Chapter 3 describes the methodology for adapting existing learned indexes to support embedded time series data and incremental appends. Implementation details are described in Chapter 4. Experimental results in Chapter 5 demonstrated improved performance of learned indexes compared to the state of the art. The thesis closes with conclusions and future work.

Chapter 2

Background

2.1 Database Systems

After the introduction of random access storage, the Relational Model popularized the notion that applications should query for data by content. This requires data to be organized in a way such that their semantic purpose was integrated into the structure. The concept of the database "table" was created to fulfill this purpose, with each table being designated to contain data of a specific type and function. A table consists of rows and columns, where each row represents a record and each column represents a field within the record. Records are also commonly referred to as tuples. The columns define the type of data that can be stored in each field, such as text, numbers, or dates. For example, a table that holds student records for the purpose of school administration might have a text column to contain the student's name, a number column that contains the student's grade number, a number column to hold the student's age, and another number column containing a unique identifier tied to the student. A populated table following this structure is shown in Table 2.1(a). The data types defined by the columns must be met for all student information to be inserted into the table. A formatted piece of information with the correct number of fields containing the correct data types as defined by the database table can then be inserted as a "row" in the database. A database row can only be inserted into a database table if and only if the row has the correct number of fields required by the table, with each field containing the correct data types. Tables within a database can be related to one another through common keys and indexes, which allow for the retrieval of data from multiple tables based on common values. Tables are the building blocks of relational databases and are used to store and manage large amounts of structured data.

Software used to manage and organize relational databases (as well as other types of databases) are called Database Management Systems (DBMS). They provide tools and features for creating, maintaining, and accessing databases, including data modeling, data manipulation, data security, and data retrieval. The DBMS acts as an interface between the database and the

StudentID	StudentName	Grade	Age
0	David	1	6
1	John	1	7
2	Joseph	1	7
3	Alex	1	6
4	Lisa	1	6
5	Allie	2	7
6	Jessica	2	8
7	Jake	2	8
8	Chris	2	7
9	George	2	7
10	Polly	3	9
11	Brock	3	8
12	Shawn	3	8
13	Jeff	3	8
14	Peter	3	9
15	Jason	4	10
16	William	4	9
17	Danny	4	10
18	Zack	4	9
19	Bill	4	9

(a) Students Table

TeacherID	TeacherName	Subject
0	Emma	Math
1	Gerald	English
2	Molly	Science
3	Harold	Social Studies

(b) Teachers Table

Grade	TeacherID
1	0
2	1
3	2
4	3

(c) Homeroom Table

StudentName	TeacherName
John	Emma
Joseph	Emma
Allie	Gerald
Chris	Gerald
George	Gerald

(d) Query Result For Age 7 Students And Their Homeroom Teachers

Table 2.1: Relational Database Tables

applications that use it, allowing users to interact with the database without having to understand the underlying data structures. DBMSs can be relational, NoSQL, or a combination of both, and they are essential for managing large amounts of data in organizations and businesses. Examples of popular DBMSs include Oracle, Microsoft SQL Server, MySQL, and PostgreSQL.

To interact with relational databases, a querying language is needed. SQL (Structured Query Language) is a standard programming language for managing relational databases. It is used to create, modify, and query relational databases. SQL is used to interact with a relational database management system (DBMS) and perform tasks such as:

- Creating and modifying database structures, including tables, views, indexes, and constraints.
- Inserting, updating, and deleting data in a database.
- Retrieving data from a database, using SELECT statements to retrieve specific data based on criteria.
- Grouping and aggregating data, using aggregate functions such as SUM, AVG, and COUNT.

– Joining data from multiple tables using join operations to combine data from multiple tables into a single result set.

SQL is a declarative language, meaning that you specify what you want to do, and the database management system takes care of the details of how to do it. This makes it a high-level language that is easy to use, even for non-programmers. SQL has been widely adopted as the standard for relational database management and is supported by a wide range of database management systems. It is widely used in various industries and has become a standard language for accessing and managing data in relational databases. Using the school administration example, we can write a specific SQL query to show the names of students and their homeroom teachers. We can then apply a filter to only show students of age 7. The full SQL query of this example is:

```
SELECT
  StudentName, TeacherName
FROM
  HOMEROOM
  NATURAL JOIN STUDENTS
  NATURAL JOIN TEACHERS
WHERE
  STUDENTS.Age = 7;
```

The result set from executing this query is shown in Table 2.1(d).

To uniquely differentiate all entries stored within a database table, each database tuple contains a unique identifier. This unique identifier is referred to as the primary key. It is used to enforce referential integrity, which ensures that all records in a database are individually addressable and that relationships between database tables are maintained. Database keys are also used to enforce constraints between tables in a relational database. These constraints ensure the accuracy and consistency of data in the database. Primary keys belonging to other tables are referred to as foreign keys. Some examples of keys are: a timestamp/datecode, a numerical ID generated by the database software, or a text field containing some unique data descriptor. On our school administration example, the TeacherID column on the Homeroom table is a foreign key that corresponds to the primary key on the Teachers table. These keys represent a relationship between the two tables, and allow them to be joined into a combined table like shown on Figure 2.1.

To store data within a physical storage medium, the DBMS must typically interact with the host operating system. The operating system then

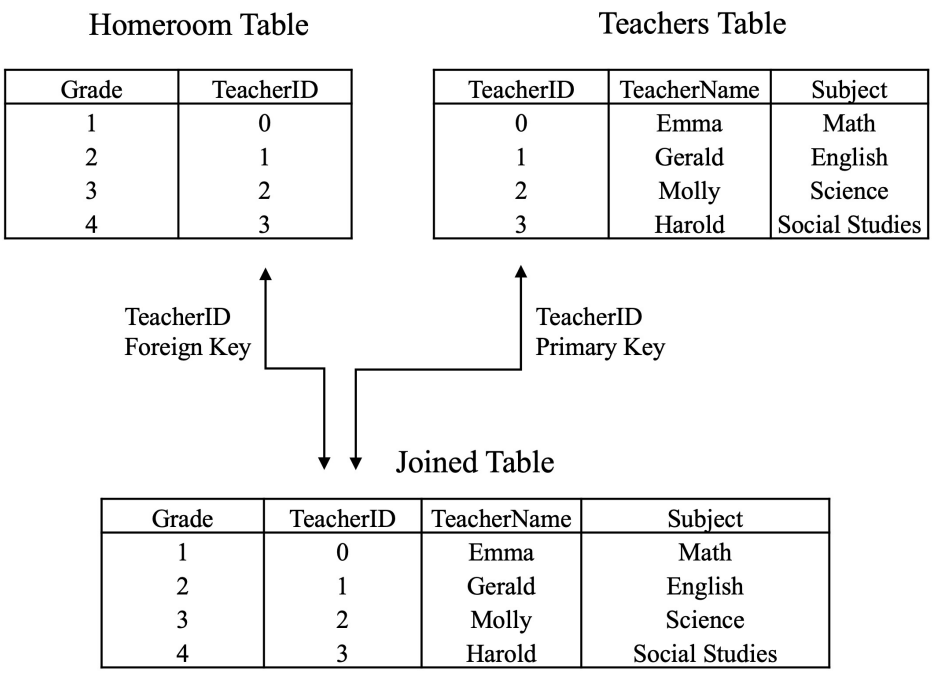

Figure 2.1: Database Keys

interfaces with the storage device, creating an abstraction layer which is illustrated in Figure 2.2. When data is stored on a disk, it is stored in *blocks* consisting of a contiguous number of bytes. Operating systems may choose to use any arbitrary block size, but it is typically between 4 KiB and 64 KiB. By using a larger block size, the operating system needs to index fewer blocks for the entire size of the storage device, which greatly benefits memory constrained devices.

2.2 Database Indexes

To improve querying and general database performance, indexes are often used in conjunction with a search algorithm to reduce the number of comparisons and disk access operations. As noted previously, storage devices must be written to in blocks. All tuples stored within the database must be distributed into these blocks before they can be written to storage. Storage blocks are not guaranteed to be contiguous and may be spread throughout the storage media. This presents problems in the context of a database ap-

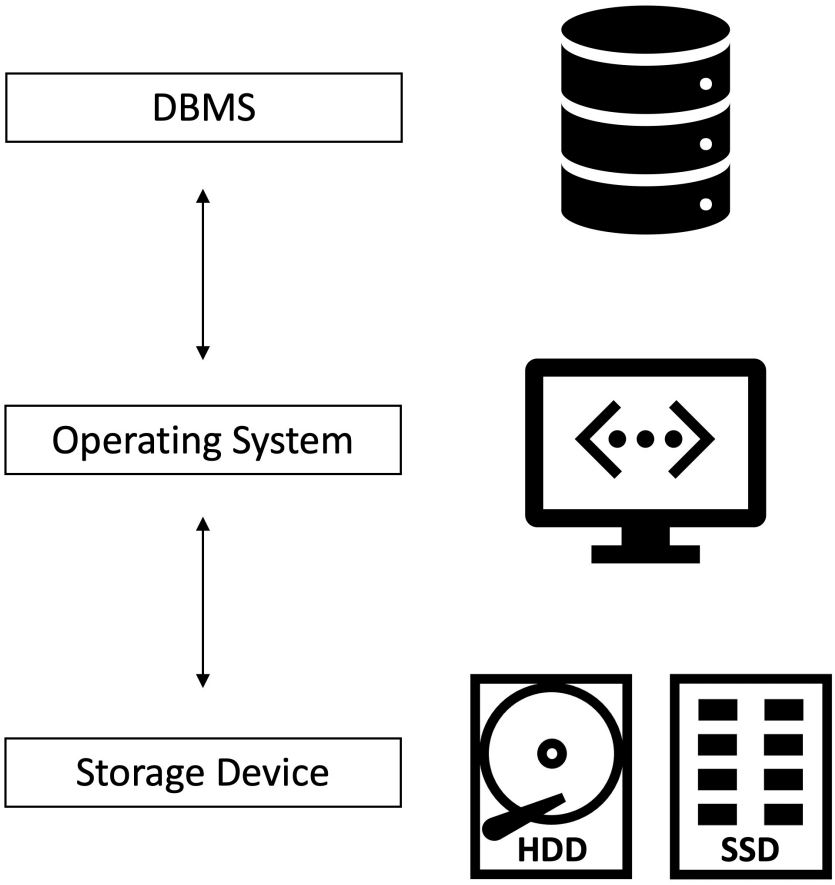

Figure 2.2: Hardware Abstraction Layers for Database Storage

plication, where a query operation searching for a specific piece of data must search through the storage device block by block until the correct query result is found. This means that to fulfill certain database queries, all the disk blocks containing database information must be read, and no search algorithm can be used. Indexes solve this issue by creating an organized data structure that stores database keys along with the corresponding pointers to the physical addresses of the data block stored on disk. This can reduce the total number of disk access operations necessary for most database operations by determining the correct pointer to the storage block containing the needed data instead of reading and searching through multiple blocks. According to [Gra93], there are several indexing approaches.

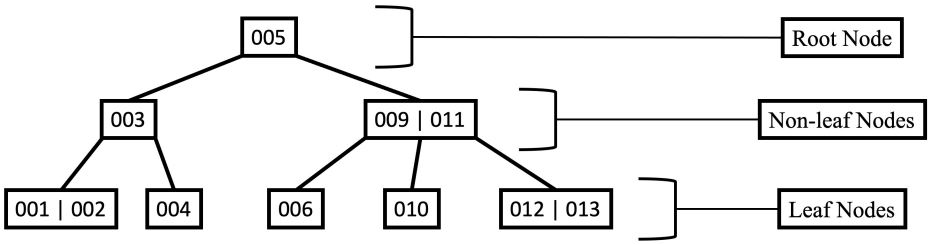

Figure 2.3: B-Trees

B-trees [Com79] are the most common and widely used database index. Many derivatives of the B-tree implementation are also in use, including B+ trees which have improved scan speed, and B* trees which have superior space utilization for random inserts. The base implementation of a B-tree is made up of several different types of nodes: internal nodes, leaf nodes, and the root node. These different node types are shown in Figure 2.3. All nodes contain index keys and pointers to the physical storage block that contains the relevant database tuple that correspond with their index key. The root node acts as the point of entry for all querying operations. This node can either be a leaf node, or have links to one or more child nodes. Child nodes can be either leaf nodes or internal nodes. Leaf nodes are nodes that do not have any children, and just contain an index key and pointer. Internal nodes are nodes that have one or more child nodes but are not the root node. These nodes contain keys known as pivots which determine the value ranges that their child nodes are split upon. On our B-tree example shown in Figure 2.3, the root node contains a single pivot with a value of 5. Consequently, this means that all child nodes on the left of the root node contain values less than 5, and all child nodes to the right of the root node contain values larger than 5. This same logic applies recursively for all nodes with child nodes to the left and right. Nodes that contain more than one pivot also have more than two child nodes. On our B-Tree example, the root node's right child contains two pivot values of (009, 011). As such, it accommodates a total of three child nodes. Left child nodes are smaller than the smallest pivot (009), middle child nodes are values between 009 and 011, and right child nodes contain values larger than 011. Nodes containing more pivots accommodate even child nodes that have values between the pivots.

Linear hashing [FOKM$^+$20] is another indexing method that aims to reduce the number of disk lookups by determining the physical address of

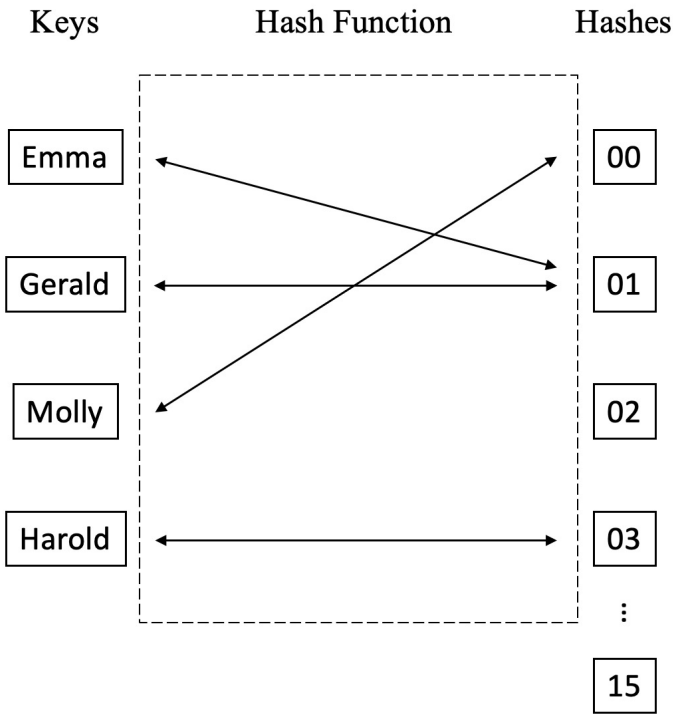

Figure 2.4: Hash index

an indexed database tuple by using a hashing method. Hashing algorithms are able to deterministically generate a hash for an arbitrary input key. This is illustrated in Figure 2.4. Note that collisions are possible where two input keys generate the same hash. When hashes are used correctly and collisions are minimized, hashing can be very fast for individual value lookups. A trade-off for this performance is that the index does not support range operations.

R-trees are another type of tree that are specially suited for multi-dimensional querying. R-trees operate by organizing data into a series of minimum bounding rectangles (MBRs), where each rectangle fully encapsulates its children with the smallest possible margins. These MBRs are then used as the pivot points in a tree structure, where coordinates can be searched according to the bounding region that the point is located in. An example of an R-Tree structure is shown in Figure 2.5, along with its spatial representation. Note that all child elements are fully encapsulated in the

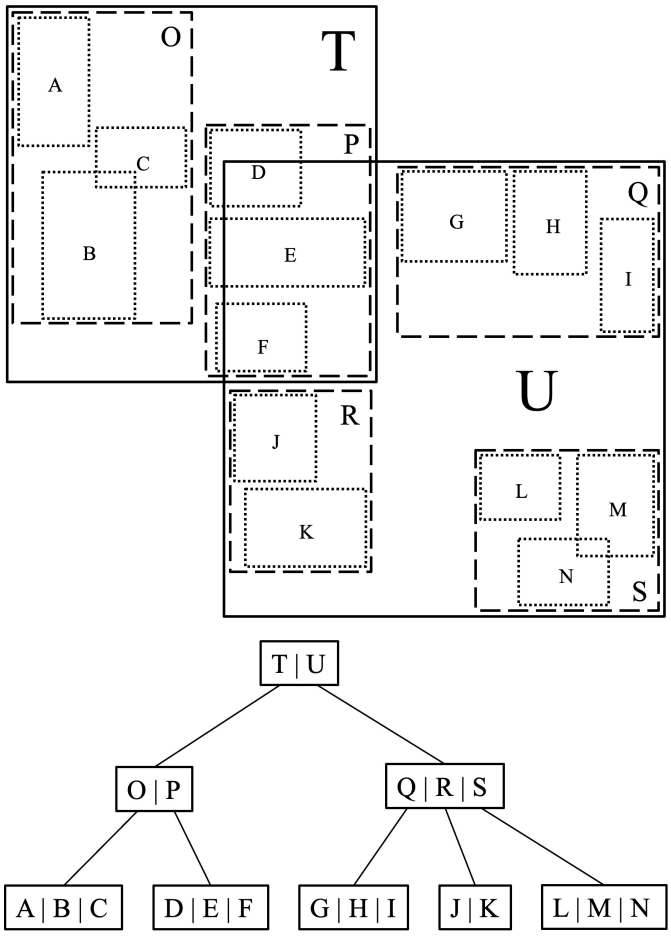

Figure 2.5: R-Tree index

MBR of the parent element.

2.3 Flash Indexes

The indexing strategies discussed thus far were developed primarily for the spinning magnetic hard disk. In recent years, Solid State Drives (SSDs) have become increasingly popular. SSDs use NAND flash as the storage medium rather than spinning magnetic disks, offering performance, durability, and power consumption advantages, but also some unique characteristics not present on magnetic hard disks [FABM19]. One such idiosyncrasy is that

pages can be individually updated on magnetic storage disks, whereas NAND memory must perform an erase and a write to complete the same operation. This is because pages stored on NAND flash cannot be directly overwritten like they can on a magnetic hard drive. Instead, groups of pages called an (erase) block must be erased before new data can be written (erase-before-write). This presents two problems, one of performance, and the other of device longevity. The performance problem is intuitive to understand, where updating a single page may require a block erase consisting of multiple pages. A update-in-place operation is extremely costly requiring reading the block of pages into memory, erasing the block, then writing back all pages including the updated page. In practice, an update to an existing page is written somewhere else in storage and a flash translation layer (FTL) used to map logical page addresses to physical locations. The longevity problem stems from the fact that NAND flash cells have a limited number of erases it can perform before the cell can no longer reliably hold data. Performing repeated updates on a single page (with each update involving an erase and write cycle) quickly wears down flash cells which can result in reliability problems. Manufacturers of SSDs know about these issues and have implemented firmware solutions to help mitigate them. These mitigations often involve writing an updated block (along with the rest of the page that the block resides in) in a new page and marking the old page as "deleted" with a tombstone marker in the drive's internal lookup table. This alleviates the immediate performance penalty of having to perform a page-wide erase but will need a garbage collection process to clear the tombstone pages when the SSD is idle. These mitigations are packaged together into the interface that manages the NAND flash on the storage drive and handles communication with the host device. This firmware is known as the Flash Translation Layer (FTL) and is able to spread the wear of the flash cells on an SSD evenly between the available flash packages instead of wearing out a single flash cell prematurely. However, it is still imperative to explore indexing methods that address the shortcomings of NAND storage, as the FTL cannot perfectly mitigate the issues associated with an indexing scheme that was originally designed to be used with a different storage technology. The existing flash indexing schemes can be broadly separated into three different categories: B-tree based indexes, skip list based indexes, and hash-based indexes.

2.3.1 B-tree Based Flash Indexes

BFTL [WKC07], IBSF [LL10], RBFTL [XYLW08]

BFTL was the first index designed for SSDs that utilized B-trees. This index combines an LSM tree [OCGO96] with a write buffer (called the "reservation buffer") to cache disk operations in the form of "IU records". LSM trees are able to batch random writes together to create blocks of sorted sequential writes before they become IU records. Each IU record contains information on the type of operation, the key used, and the tree nodes affected. When the reservation buffer fills up, IUs are organized by the nodes they affect, then written to NAND. This organization process may result in IUs from different nodes sharing a NAND page. Thus, this index uses an accompanying Node Translation Table (NTT) to reconstruct the tree structure from the data contained on the NAND. Improvements to BFTL such as IBSF and RBFTL employ better buffer management schemes to avoid splitting up nodes into different flash pages, and in the case of RBFTL, employs the usage of a small NOR flash cache to backup IUs before inserting them into the buffer.

Lazy-Update B+tree [OHLX09], MB-tree [RKKP09], uB+tree [Vig12], PIO B+tree [RPSL14]

These indexes also aim to delay update operations by buffering them in memory using a tree structure. Lazy-Update B+tree achieves this by buffering updates using the leaf nodes of a tree structure. This process is resource intensive, requiring CPU and memory resources to repeatedly query the tree nodes. MB-Tree attempts to address this by using fat leaves on the tree structure. These fat leaves require the use of additional nodes, which also needs hardware resources to query. The uB+tree (short for unbalanced tree) forgoes the tree balancing operation for as long as possible, thereby saving on write operations. Tree nodes are assigned additional overflow nodes as needed, until they are read a specified number of times, at which point the nodes are finally split and balanced in a regular fashion. The PIO B+tree further improves performance by batching multiple IO operations to perform at once, exploiting the parallelization capabilities of modern SSDs.

FD-tree [LHY+10]

FD-tree has the additional capability to transform many small random writes into sequential reads and writes. As sequential reads and writes are

much faster than random reads and writes, this transformation has significant performance advantages when applied to SSDs. However the FD-tree has only been experimentally validated, and it is uncertain whether it can be used on real systems.

BF-tree [AA14]

Utilizing bloom filters, the key-memberships within B-tree nodes are recorded and stored alongside the values. This has good search performance, with the trade-off of costly insert performance. As such, it is designed to be used as a bulk index, with little flexibility for updates.

AB-tree [JWZ+15]

AB-trees employ the use of buckets that serve as storage structures for data, and buffers for IO operations. This allows delete, insert, and update operations to be processed in batches when a buffer fills. The size of the buckets and buffers within each tree node can also be sized dynamically according to the workload being conducted.

FB-Tree [JRvS11]

This index tree performs all updates out-of-place, appending the updated block in its entirety to the end of the file. As such, the newly written blocks do not have space to accommodate additional updates. This may reduce the overall lifespan of the SSD as it is less write-efficient for update operations.

Flash B-Tree Indexes Summary

B-Tree indexes designed for flash memory aim to reduce or eliminate the in-place updates used by conventional B-Trees, as these operations are inefficient on flash-based storage devices. The BFTL, IBSF, and RBFTL indexes accomplish this using a write buffer to cache disk operations before they are written to NAND as IU records. An accompanying Node Translation Table is used to reconstruct the tree structure from the stored IU records when the B-Tree needs to be read.

– The Lazy-Update B+Tree, MB-Tree, uB+Tree and PIO B+Tree buffer update operations in memory, grouped together according to the leaf nodes that they affect. When the buffer overflows, a policy is used to select groups to be written to disk such that overall disk writes are reduced.

- The FD-Tree uses a separate structure to handle updates, which is used to turn random writes into sequential writes to improve performance.
- The BF-Tree utilizes bloom filters to record key-memberships, exhibiting improved search performance at the cost of insertion performance.
- AB-Trees use buckets in-place of nodes to batch NAND update operations. The size of the buckets can be sized dynamically according to the workload.
- FB-Trees write all updates to the end of the file, which has good throughput performance but may reduce the overall lifespan of the storage device, as the newly written data pages do not have space to accommodate additional updates.

2.3.2 Skip-List-Based Indexes

Skip-lists operate using a probabilistic model, where the most commonly accessed regions of the list have "shortcuts" that make them easily accessible without needing to go through an extensive search process. Skip lists are composed of multiple levels, where the base level is simply a sorted linked list of nodes containing all data elements. Levels built on top of the base layer are also sorted linked lists, the difference being that nodes within upper lists contain additional pointers to other nodes from a list in a lower level. All pointers to nodes in lower lists are accompanied by a key containing the lowest-valued element of the node being pointed to. These keys are referred to as "pivots" as they divide the lower levels into discrete search ranges. Upper levels of a skip-list "skip" multiple entries in the layer below it to allow for quick traversal to the most commonly accessed entries, skipping over the entries that rarely get accessed. For example, the upper level on a two-level skip list with 5 elements may choose to only have pointers to the 3rd and 5th element on the base level list if the 2nd and 4th elements are rarely accessed. Policies on the construction of the skip list layers vary between implementations, some examples of which are listed below.

FlashSkipList [WF17]

FlashSkipList uses an in-memory buffer to store incoming write operations. When the buffer overflows, it is flushed to flash where it is dynamically associated with an element in a skip-list search structure. An online competitive algorithm adjusts the distribution of the write buffers within the

skip list whenever an upper size bound is reached, or a search operation is conducted. FlashSkipList has been experimentally demonstrated to deliver superior performance than the BFTL, FlashDB, LA-Tree and FD-Tree indexes.

Write-Optimized Skip List [BFCJ+17]

Like the FlashSkipList, the Write-Optimized Skip List implementation also uses in-memory buffers to store incoming write operations. However, the Write-Optimized Skip List allocates additional space inside its buffers inside the nodes of the skip-list structure for insert operations. Nodes also contain a write buffer which splits its elements into buffers on lower levels upon overflowing, while a new pivot is inserted into the layer above. The performance of Write-Optimized Skip List has only been theoretically analyzed, real-world performance remains un-tested.

Flash Skip-List Indexes Summary

Skip lists are able to use probabilistic models to optimize search queries to the most commonly accessed entries. Both FlashSkiplist and Write-Optimized Skip List have unique policies to determine how pointers within the dataset should be organized, but Write-Optimized Skip List also incorporates a buffer area within the data nodes to accommodate additional data inserts.

2.3.3 Hash-Based Indexes

Hash-based indexes are built around the hash table data structure, where a hashing algorithm is able to evenly and deterministically distribute an incoming stream of keys into a collection of data buckets. Pointers to the data buckets and the hash value they are associated with are organized in a table termed as the "directory". When searching for a value, the key is evaluated with the hashing algorithm, where the returned hashed value is referenced using the directory to locate the bucket containing the value associated with the un-hashed key. This search structure is very read efficient, as it can return a specific pointer to the data bucket containing the desired search value often in only one access. There are several types of hash tables including linear hash, extendable hashing, and bloom filters. Linear hashing handles bucket overflows by linking additional overflow buckets from the base bucket to store overflow value entries. By comparison, extendible hashing does not use overflow buckets. Instead, overflows are handled by increasing the size of

the directory so that it can point to a new bucket to contain overflows. This ensures that extendible hashing will have a O(1) worst-case access time, the trade-off being that the directory will have a super-linear size. The bloom filter data structure uses multiple hashing functions to generate a combined, compound hash value. The number of hashing functions used is determined by the desired overall error rate, since the bloom filter is able to answer set membership queries with a chance for false-positives but zero false negatives. This results in a very space and time efficient index. In the context of flash memory, operations like inserting values and bucket splitting often involve in-place updates and random writes which are detrimental for SSD performance and longevity. The following are a selection of flash-aware hashing indexes that attempt to remedy this problem.

LS Linear Hash [LDM08]

LS Linear hash handles bucket splitting by constantly monitoring search performance, and only performing the bucket splitting operation when performance levels fall below a threshold. Bucket splitting is then done in batches that optimize writes in return for increased reads.

HashTree [CJY10]

HashTree uses a bounded-height FD-tree (a B-Tree structure designed for flash) as storage buckets. The sorted values in the FD-Tree can then be written to flash with sequential writes rather than random writes.

SAL-Hashing [JYW+18]

The buckets in SAL-Hashing are arranged together into units that are batched together and written to disk simultaneously, taking advantage of the fast parallel IO characteristics of SSD storage. This scheme introduces an online algorithm to split and merge logs into buckets, which introduces a penalty in search performance.

BBF [CMB+10], BloomFlash [DSl+11] and FBF [LDD11]

Both BFF and BloomFlash are bloom filter derivatives that use similar approaches to incorporating bloom filters on SSD storage. Bloom filters are split up into smaller sub bloom filters that are exactly one page in size, such that write operations can be done without conducting any in-place updates. A limitation of both BBF and the BloomFlash index is an upper bound

on the number of elements that can be stored. FBF solves this issue by organizing the sub bloom filters hierarchically such that when a level reaches capacity, a new level is generated wherein each element in the new level incorporates a predetermined number of blocks from the previous level.

Flash Hash Indexes Summary

Hash based indexes use hashing algorithms to split up incoming keys into a collection of data buckets.

- LS Linear hash defers bucket splitting until performance falls below a threshold.
- HashTree uses B-tree structures as buckets to speed up querying performance.
- SAL-Hashing groups buckets together to be written to disk simultaneously. An online algorithm determines how to split and merge write logs into buckets.
- BBF and BloomFlash utilize bloom filters to index pages stored on flash. The limitation of both these indexes is the an upper bound on the total number of elements that can be stored.

2.3.4 Flash Indexes Performance Summary

This section summarizes the performance characteristics of the indexes discussed along with their space efficiency (if available).

Tree-Based Indexes

Index	Search	Insertion/Deletion	Space
BFTL	$h * c$	$2(\frac{1}{M-1} + \tilde{N}_{split} + \tilde{N}_{merge/rotate})$	$n * c + B$
IBSF	h	$\frac{1}{n_{iu}}(\tilde{N}_{split} + \tilde{N}_{merge/rotate})$	$N + B$
RBFTL	–	–	–
Lazy-Update B+Tree	–	–	–
MB-Tree	$2 + \lceil log_M \frac{2*n}{M*n_l} \rceil$	$[3/n_f]_w + [(n_l + \lceil \frac{2*n}{M*n_l} \rceil)/n_f]_r$	$N + B$
uB+Tree	–	–	–
PIO B+Tree	$h - 1 + t_L$	$[\sum_{l=\lfloor \eta \rfloor}^{h-2} \frac{1}{G(l)} - \frac{1/M'^{(\eta \% 1)}}{G(log_{M'}(\mu - B) - 1)} + \frac{1}{G(h-1)}]_r + [\frac{1}{G(h-1)}]_w$	$n + \mu$
FD-Tree	$log_k n$	$[\frac{k}{f-k} log_k n]_{srw}$	$c * n$
BF-Tree	$h + [p_{fp} * n_{pl}]_{sr}$	–	$n * n_{pl}$
AB-Tree	$\sum_{l=1}^{h} \frac{M'^{l-1}}{N_l} l$	h/s_n	n
FB-Tree	–	–	–

Skip-List-Based Indexes

Index	Search	Insertion/Deletion	Space
FlashSkipList	–	–	–
Write-Optimized Skip List	–	–	–

Hash-Based Indexes

Index	Search	Insertion/Deletion	Space
LS Linear Hash	–	–	–
HashTree	–	–	–
SAL-Hashing	$2 + n_{lp} * p_{fp}$	$\frac{(r+4)*g_n}{B*s_p - 2*g_n} * (2 + \frac{g}{n} + p_u * g)$	n
BBF & BloomFlash	$1/n_{bf}, 1$	$1/n_{bf}$	$n + B$
FBF	$log_b n$	$1/B$	$n + B$

Table 2.2: List of flash indexes and their performance [FABM20]. – denotes a lack of available data, the r, w, s, bm subscripts stand for reads, writes, sequential and block merges respectively. Refer to Table 2.3 for a list of symbol definitions.

Symbol	Definition
b	Number of child structures introduced to each block after false positive limit is reached
B	Size of reservation filter
c	Predefined threshold before tree reorganization
f	Number of entries per page
g	Number of buckets in a group
g_n	Number of groups
$G(i)$	Average number of update operations that read a node at level i
h	The height of the tree
k	Logarithmic ratio between the size of adjacent levels
l	Variable that denotes a specific level in a tree
L	Size of leaf
M	Number of keys per node in the context of B-Trees. Number of IUs (write logs) per flash page in the context of hashing.
M'	Average number of entries in a node
μ	Amount of available memory
n	Number of nodes
n_{bf}	Number of buffered operations
n_f	Number of flushes
n_{iu}	Number of buffered IUs
n_l	Number of leaves
n_{lp}	Number of pages a log area
n_{pl}	Number of pages per leaf node
η	Expands to $log_{M'} \frac{n}{L(\mu - b)} - 1$
N	Number of keys
$\tilde{N}$	Number of operations denoted in the subscript
p_{fp}	Probability of false positives
p_u	Probability that a particular record resides in a particular log area
r	Record size
s_n	Average size of a node
s_p	Page size
t_L	ηTime to read a leaf of size L

Table 2.3: List of Symbols Used in Table 2.2 and Their Definitions

2.4 Time Series Databases

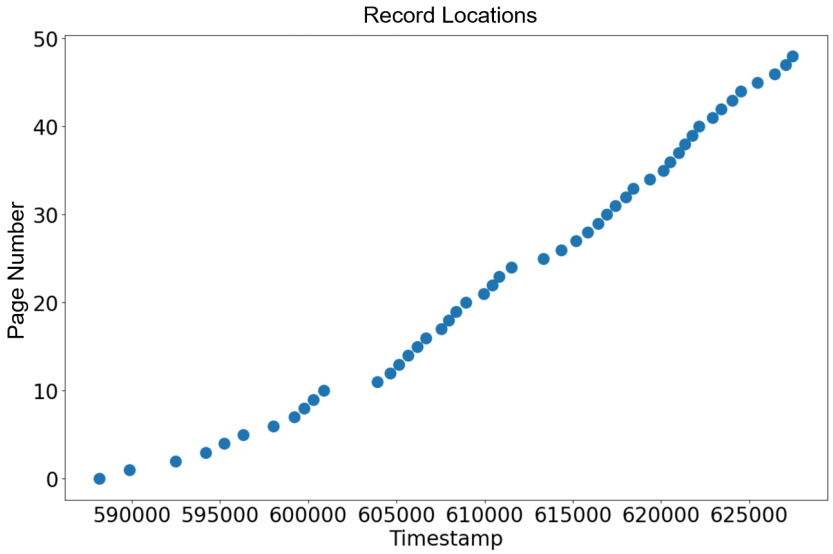

Figure 2.6: Sample Time Series Dataset

A time series dataset consists of a number of key-value tuples, with a timestamp as the key and a piece of arbitrary data as the value. These timestamps are stored in ascending order such that the data represents a temporal progression of values over time. Each data recording is assigned the current timestamp and inserted into the time series dataset in the order of which they are collected. An example of a time series dataset depicting data page numbers that contain information pertaining to specific timestamps are shown in Figure 2.6. Note the increasing and sorted order of the timestamps. Tuples of time series datasets are stored on-disk, with each physical page on the disk containing multiple tuples. Figure 2.7 shows the general data hierarchy of how tuples are stored on disk. Many database systems designed to contain time series data exist and have different approaches on how best to store time series data. Some examples are explored here.

2.4.1 LittleTable

LittleTable [RWW+17] is a time series database system used by Cisco Meraki to store usage statistics, event logs, and other time-series data collected from customer devices. This data is used to generate data visualiza-

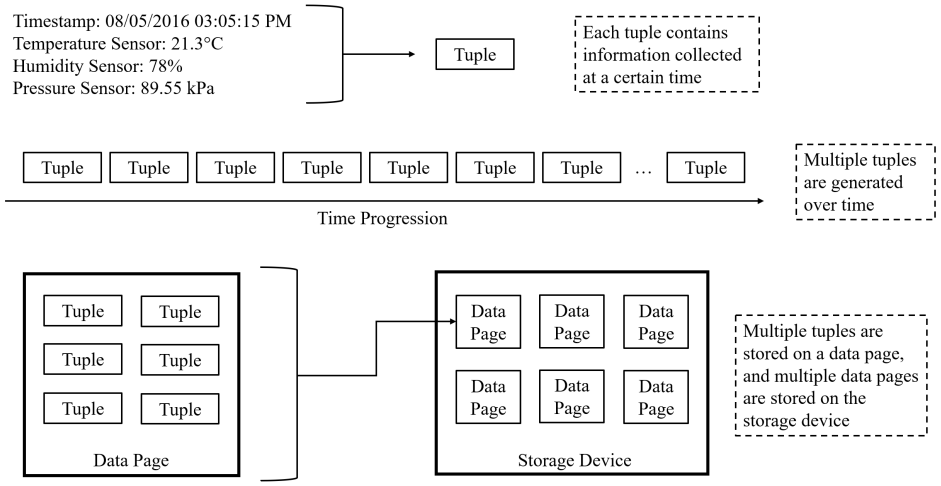

Figure 2.7: Data Organization of Tuples on Disk

tions like bar graphs and tables displayed with an online dashboard, but can also be used to perform diagnostic tasks like root-cause analysis and data forensics. From the database design perspective, this means that:

- Frequent queries retrieving the same data must be very fast. This is because the most common data visualizations only request the most recent data (e.g. The number of bytes transferred by a client in the last hour, rankings of top network users of the past month, etc.).
- Queries retrieving older data must also be quick, as data forensics and root-cause analysis applications benefit from the ability to look back further in time.
- The database system should be optimized for use with magnetic disk hard drives as the storage medium. Storing a longer history of logs takes up a larger amount of space, so magnetic disks are preferred over solid state media for economic reasons as the latter still has a considerably larger cost per byte than magnetic disks.

Taking these considerations into account, LittleTable was designed using a B-tree indexing system that iteratively adds new records in chunks called 'tablets' as they are recorded. As with any time series, the keys are timestamps sorted in ascending order. LittleTable intelligently combines a device ID with the timestamp to form a new key, such that devices that are related

to each other (eg. in the same network) have contiguous keys. This has the result of making it likely that data for related devices are also stored on contiguous blocks on disk. Since data for related devices are often requested simultaneously, the hard drive does not need to incur a seek penalty to read data for related devices. As new data is inserted, it is first inserted into a tablet. Each tablet contains a B-tree index to speed up queries and reduce operational latency. This index only takes up 0.5 percent of the total size of the tablet, and is cached on main system memory. The minimum and maximum timestamps of each tablet is also cached on system memory and is accessed using binary search. This means that lookups of newer data takes roughly the same amount of time as lookups of older data. Cisco runs LittleTable using powerful servers, which they call 'shards'. Shards are able to rely on system memory to cache index data and speed up data queries, but on lower power devices with limited memory, a different approach may be required.

2.4.2 Apache IoTDB

Apache IoTDB [WHQ+20] is a time series database developed to address the shortfalls of NoSQL when working in the context of Industrial Internet of Things (IIOT). The increasing popularity of IoT devices motivated the creation of a time series management system optimized for high-throughput and low-latency with support for common operations used in time series analysis. Additionally, IoT devices are diverse in their implementation, from consumer applications like wearables, smart-home devices, to industrial and enterprise applications like connected healthcare and monitoring. This wide gamut of applications necessitates the creation of scalable and portable software with wide hardware support. For this reason, IoTDB supports both cloud and edge deployment models. On the cloud side, IoTDB can be deployed in distributed computing clusters housed in data centers. On the edge side, IoTDB can also be scaled down to be hosted on a standalone workstation PC, or even on an embedded edge device like a Raspberry Pi.

Regardless of the deployment model, IoTDB has been optimized for particular use cases and features most relevant for IoT devices, namely:

- Edge computing. With the capabilities of edge devices quickly rising in recent years, the practice of conducting data analysis on the edge-side has gained prevalence. This requires IoTDB to be run on both the cloud side and the edge side, with an organizaiton structure that supports data synchronization between the two.

- Long-life and large volume historical data. Industrial IoT applications can generate a large amount of data that must maintain high availability for extended periods of time. A Boeing 787 jetliner was used as an example in the original publication. Its monitoring sensors can produce more than 500 GB of data over the course of a single flight. Additionally, long historical data logs must be maintained to identify trends and monitor the life-cycle of safety-critical components.

- High throughput data ingestion. With the large volume of data being generated by industrial IoT devices, high throughput data ingestion is required to absorb all the data without dropping data samples.

- Low latency and complex queries. Applications that require real-time monitoring require low latency query fulfillment to quickly identify potential equipment faults. Additionally, deeper delves into the data by data-scientists can be complex and require flexibility on the types of queries that must be supported.

- Advanced data analytics. Advanced data analysis techniques like machine learning must compute certain metrics after processing chunks of historical data using a costly ETL (Extract, Transform, Load) method. The common operations used in such data analysis techniques must be done efficiently.

To accommodate these design goals, IoTDB was designed with a uniform design across the edge and cloud implementations which allow it to be run on ARM7 processors with as little as 32MB of memory. Data can also be easily synchronized with cloud instances of IoTDB using a File Sync module. Support for long life, high throughput, low latency and advanced data management is handled using a combination of the TsFile data format and intelligent indexing methods. The TsFile format is composed of two primary components, the data content, and the index. The data portion is partitioned into pages, which are the smallest storage unit on disk. These pages form a chunk of data, with each chunk containing time series data for a specific time range specified by the index portion of the file. Additionally, a *Summary Info* section is present in each chunk of data to store commonly used aggregate information such as the max/min values and timestamps of each chunk, the average value, data count, etc. These commonly used metrics are pre-calculated at the time of insertion and greatly improve performance to enable complex, low latency querying over a large pool of data. Additionally, IoTDB uses write-ahead logging to facilitate high write throughput

performance. This streamlines the time-ordering process, and leverages the faster sequential write performance of the storage device. IoTDB's design lends itself greatly in applications where flexibility of data retrieval and performance is the primary focus. In applications where embedded devices are deployed on battery power, energy consumption becomes a very important metric to optimize to prolong the lifespan of the device. For these scenarios, alternatives to IoTDB are worth consideration.

2.5 Embedded Databases and Sensor Networks

According to Wolf [Wol02], embedded systems are devices which act as a component in a larger system, and rely on its own CPU. Embedded CPUs also require their own memory, which it uses to perform a predefined task as part of a larger process. This decouples hardware design and software design, allowing for the same software to be reused in future products that perform similar functions. As electronic devices become more sophisticated and perform more functions, they incorporate more embedded devices to facilitate the added functionality. Common examples of embedded devices that are part of a bigger whole include:

- The Digital Signal Processors (DSPs) on mobile phones that translate radio signals into digital data signals, or convert digital data signals into analogue wave forms that can be perceived as sound.
- The engine management computers in internal combustion vehicles which control spark and ignition timing to ensure optimal fuel burn and increase fuel efficiency.
- The co-processors found in inkjet printers that handle typesetting and enable real-time control of the print head, so that pulses of ink are applied at the exact moment the print head reaches the correct point on the page.

Embedded devices can also be networked together wirelessly to form a larger collection of devices that are able to exchange data with one another. These devices are often used in locations where direct communication links are impracticable, or impossible all-together. This means that these devices often run on battery power or scavenged energy, and have all the resources they need to function semi-autonomously. By attaching a sensor to these devices, a sensor network is created.

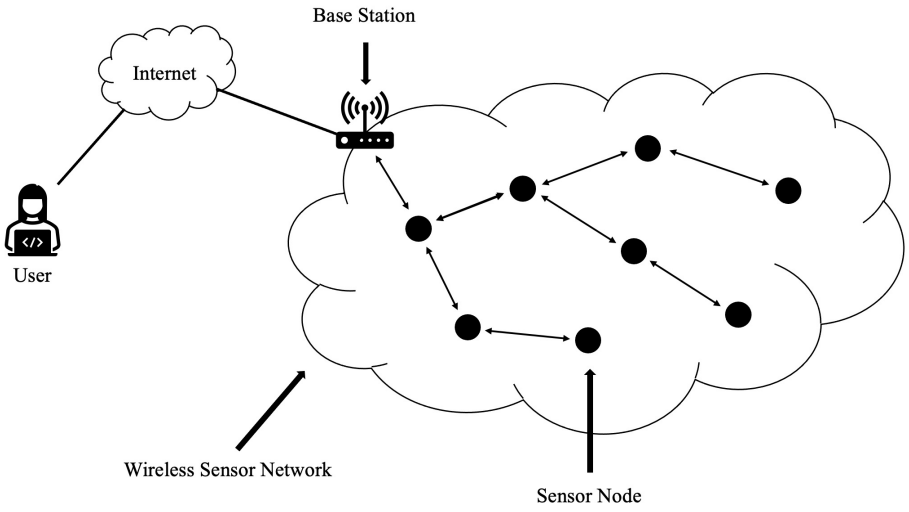

Figure 2.8: Wireless sensor network

Sensor networks are used in scenarios where data must be collected over an area larger than what a single sensor can reliably service. In such scenarios, a collection of networked sensors must be distributed to sufficiently cover the area with their combined sensor ranges. These sensor networks are often used in applications such as environmental monitoring, surveillance, and traffic management. Individual sensor nodes are equipped with capabilities to observe and record sensor data and are often run by very small and efficient embedded microprocessors. All sensor nodes communicate with a base station to report their sensor readings, where it is aggregated into a centralized database. Figure 2.8 shows an example sensor network containing a collection of nodes and a base station. During operation, nodes gather data from attached sensors and must store it locally or transmit to the base station. The storage and transmission actions are candidates for optimization, as performing such actions are costly in terms of energy consumption. In the context of a battery powered device where the total available power is limited, energy consumption becomes the forefront of priorities of things to be optimized in service of prolonging the service life of sensor nodes without requiring user maintenance. Examples of successful wireless sensor networks deployments include the Great Duck Island project [DB10], in which underground sensors were used to monitor the nests of storm petrels where they collect health data and relay the information to a gateway node

placed nearby. They have also been used as a safety tool in the mining sector, where they monitor the surrounding air quality for traces of hazardous gases to minimize the risk of explosion, contamination, and inhalation injury [CFT07].

The three types of sensor data networks defined by Tsiftes and Dunkels [TD11] are as follows.

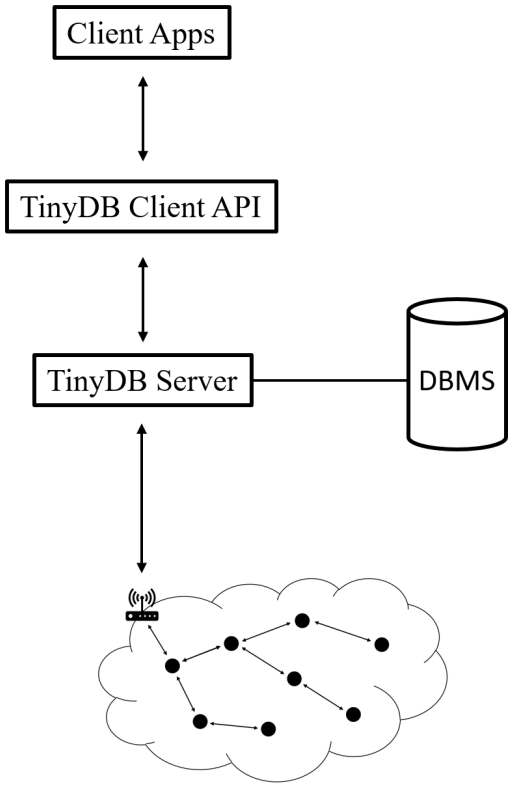

Figure 2.9: TinyDB (a data collection architecture)

Data collection networks were one of the first sensor network architectures to be introduced. In this type of sensor deployment, all sensor nodes submit their data to designated data sinks, using a data collection protocol. Though data collection networks theoretically can support data aggregation, it is not trivial to implement and as a result the feature is rarely used in practice. A notable example of a data collection network architecture is TinyDB shown in Figure 2.9. This architecture streams data from sensor nodes behind an

abstraction layer that orchestrates the data flow around sensor nodes. To the user, only an SQL-like database querying interface from a designated data gateway node is exposed.

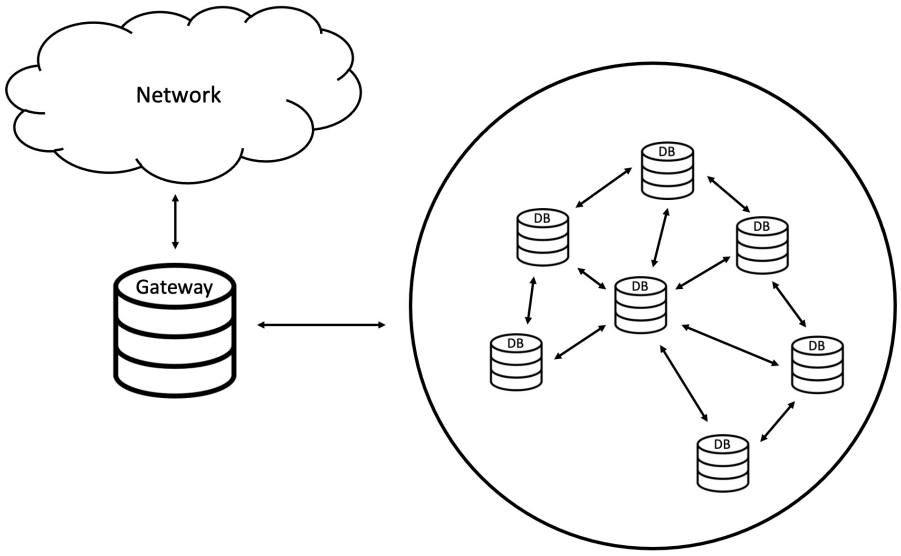

Figure 2.10: Data logging architecture

Data logging networks are another prominent sensor node architecture where all sensor nodes log their readings onto a bank of onboard storage to be retrieved in bulk at a later time. The Golden Gate bridge network is an example of a successful deployment of this type of sensor network. In this example, the network was outfitted with 40 sensors to monitor bridge conditions. The stored sensor data is collected regularly using the Flush data transfer protocol. Data logging networks are suitable in cases where the complete dataset must be gathered from all nodes. The top down view of a data logging network is shown in Figure 2.10

Data mule networks (also called disruption tolerant networks) aim to operate under circumstances where a reliable network is not available. In this network architecture, data transfer is achieved by first uploading the data onto an interim storage device (sometimes referred to as data mules), which is then physically moved into an area where network access is possible. The data is then offloaded from the storage device onto the network where it is finally routed to the collection point. The top down view of a data logging network is shown in Figure 2.11. This sensor architecture is

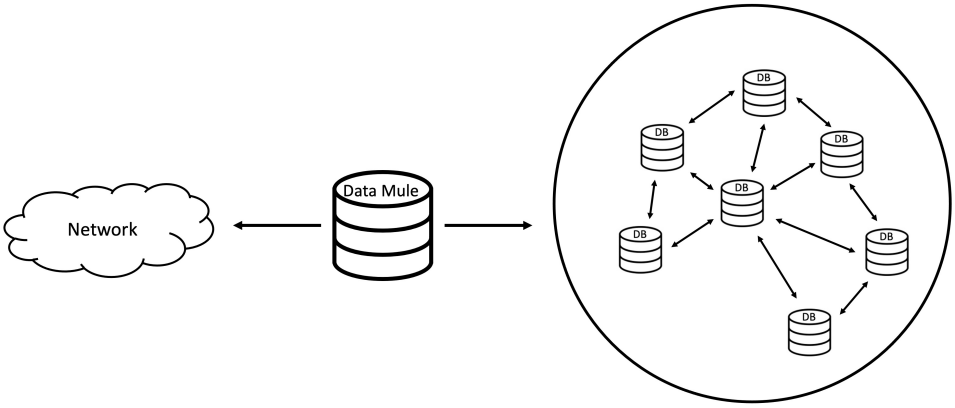

Figure 2.11: Data mule architecture

useful in remote or rural areas where network infrastructure is sparse, or in disaster areas where the network infrastructure is offline. The trade-off for this type of sensor network is the significant latency introduced by needing to physically move the data mule between the data collection site and the network connection site.

When considering the energy cost of sending wireless transmissions, sensor network architectures are optimized to reduce the usage of the onboard radio to save power. In the case of the data collection architecture, the user initiates the data collection command, and thus triggers the transmission of sensor data into the designated collection point. In this architecture, the data transmissions are triggered on demand and is thus difficult to optimize. On the other hand, the data logging and data mule sensor network architectures operate by recording sensor data onto a local storage device as they are collected, then sending the data in bulk at predetermined intervals. These architectures retrieve data from their storage devices at the time of data transmission, and use indexing methods to ensure that the retrieval operation happens in a timely manner. By introducing more efficient indexing schemes for embedded devices, sensor networks are allowed more flexibility on their data transmission intervals. This has the potential to allow for further network optimization, like longer sampling intervals between data transfers to lower energy consumption and extend the operational lifetime of the sensor network.

2.6 Design Considerations for Embedded Sensor Networks

Tsiftes and Dunkels [TD11] lay out the technological considerations that must be factored into the design of an embedded wireless sensor network deployed in a real-world operation. Firstly, energy resources of the embedded device are limited. This is the result of powering such embedded devices using batteries, which have limited energy reserves. In most cases, the lifetime of the sensor network is directly determined by the amount of energy reserves available to the device. In practice, such small, embedded devices are often placed in difficult to service areas, so battery replacement is seldom done. Therefore once the onboard energy reserves have been depleted, the device becomes inoperable. With this in mind, system designers choose hardware that minimizes power draw to increase the battery life (and thus the lifespan) of the device. Even in scenarios where energy is captured continuously from the environment, such energy sources often have low power outputs that still constrain the power draw of the embedded device. For devices that are powered from a stable power source, the power consumption of such devices directly affect the operational cost of keeping a deployment of such devices active. For example, in a deployment of a million devices, the small individual power consumption of the device all add together to form a considerable cost that must be weighed against the benefits from operating the fleet of embedded devices. This limitation on power draw also limits the type of hardware that can be powered. High performance CPUs and large pools of system memory have elevated power consumption compared to their low-power counterparts, meaning that embedded devices are often heavily constrained in terms of CPU, memory, and IO. This hardware limitation has ramifications on the querying operations and indexing schemes that can be successfully deployed. Historically, database software have first been developed for usage on the enterprise side, running on powerful servers which provide the performance needed to satisfy enterprise customers. These enterprise software packages are then adapted down to run lower power devices, but often they inherit some of the low-level design decisions made to optimize performance on large powerful machines at the detriment of performance on lower power machines. Examples of which include memory hungry hash indexing schemes which are incredibly fast on powerful machines with ample access to memory, but are not viable on low power embedded devices without such resources. Secondly, the bandwidth to communicate with the embedded device is also limited. The transfer speed to and from the embedded device

is tied to power consumption, where higher transmission rates require more power. This is due to the higher modulation and demodulation speeds required, higher clock speeds and wider bus widths of the onboard electronics, which all contribute to higher energy usage. Though some optimization is possible, the general trend of higher energy consumption hardware remains. Thirdly, the storage capabilities of the device is functionally infinite. Contrary to the restricted availability of power and communication bandwidth, the storage capabilities of an embedded device is not a limiting factor. Storage devices are a static component, and as such, larger capacity storage devices still fit into the same standardized form factor, and do not incur any increased energy cost. The proliferation of cheap and large storage devices means that embedded devices effectively have unlimited storage space for record keeping. Modern flash devices have enough capacity to fit the entire dataset an embedded device is expected to generate within its lifetime. For example, an 8-byte value recorded every second for a period of 10 years will only be 2.5 GB in size. This dataset can easily be contained by a commodity 4GB SD card, which can be purchased for less than a few dollars. It is also worth mentioning that the comparative power usage of writing data to storage is much lower than the power usage of using the radio to send the same data to a neighboring node. The experiments done by the Antelope team [TD11] show that the energy cost of sending a 100 byte packet of data is 20 times more than the energy cost to write the same data onto a flash storage device. In summary, embedded devices exhibit the following characteristics:

- Extremely energy constrained. Embedded devices are often battery powered, which limits the amount of power that the device can draw. In many cases, the service life of the battery also determines the service life of the device. This means that energy efficiency is paramount to prolong the operating life of the device.

- CPU and memory constrained. The limited power available from the battery also limits the processing capabilities and the memory amounts that can be put onto an embedded device without negatively impacting efficiency.

- Bandwidth is limited. Wireless transmission speeds are also tied to energy consumption. Faster switching and signal modulation requires higher transmit power and more powerful electronics.

- Storage is functionally infinite. Advancements in flash memory have made commodity flash storage cheap, reliable, and dense. Inexpensive,

off-the-shelf SD cards are available in capacities that have more than enough space to fit the entire dataset an embedded device is expected to generate over its lifetime.

2.7 Embedded Sensor Network Database Architectures

Several database systems designed for use on sensor networks have been developed and have been successfully deployed in commercial operations. The design of these systems are mindful of the hardware and energy constraints that characterize the embedded devices which form the sensor network nodes.

2.7.1 TinyDB

The TinyDB sensor network [MFHH05] popularized the idea to think about sensor networks as a database to be queried rather than a set of basic sensors whose only function is to relay information to a collection station, where all the data processing happens. TinyDB advocates for Acquisitional Query Processing (AQP) where data is retrieved as needed whenever a query calls for it. TinyDB introduces a query processor that functions as a front-end interface for developers to interact with the sensor network without needing to query individual sensor nodes manually. The query processor also implements filtering and aggregation capabilities within the sensor nodes to minimize radio usage and thus improve energy efficiency.

2.7.2 Antelope

In the Antelope sensor network model [TD11], each individual sensor node contains a database supporting dynamic querying and continuous data insertion. Contrary to many sensor network models that came before it, Antelope stores data on sensor nodes which can be used to optimize data collection operations. This is useful in sensor network use cases that do not rely on real time data to be sent at the time of querying, and instead aggregate data from multiple rounds of sensor samples together. This process of data aggregation where multiple data samples are sent to the head node at the same time has been demonstrated to reduce network usage and thereby lower the power consumption of the sensor node [TD11]. The data stored inside the sensor nodes are not only limited to storing sensor readings. They can also be used to store network performance metrics and routing tables

that can be used to further optimize data transmission throughout the sensor network. Antelope also introduces a SQL-like language called AQL to query the stored data. Antelope has three types of indexes that it uses, depending on the scenario. A novel MaxHeap index designed for use on NOR flash, an inline index that makes use of a linear function, and a hashing index. Additional details of these indexing methods can be found in Section 2.8.

2.7.3 Cougar

The Cougar sensor database system [BGS01] was developed to integrate data taken from embedded sensor devices into a larger relational database. This is achieved through the use of Abstract Data Types (ADTs), with every ADT type representing a specific type of sensor (temperature, humidity, seismic, etc), and every ADT object representing a physical sensor in the real world. These ADTs are used to perform signal processing on the sensor node, and transmit the resulting data back to the base station. Virtual relations are also used to encapsulate the results of ADTs along with the timestamps at which they were returned. This allows for data acquisition and processing to happen asynchronously, alleviating concerns of data collisions when collecting readings from multiple sensors at once, and also removes the latency associated with wireless data acquisition.

2.7.4 LittleD

LittleD [DL14] is a database system designed specifically with low power embedded devices in mind. Like the database used in the Antelope sensor network model, the individual IO operations for the underlying memory is abstracted. Developers are able to interact with the database using a querying language instead of needing to manually access and process data programmatically. Where Antelope introduced a bespoke querying language called AQL, LittleD implements a SQL-compliant query engine. While AQL is simple to learn and can streamline certain database operations, it has some restrictive query processing limitations that limit the type of queries that can be executed, and does not benefit from the ubiquity and standardization of SQL. LittleD has also been demonstrated to use significantly less memory than Antelope when running a benchmark of queries, with Antelope using at least 500% more memory on intensive queries [DL14]. The drastically lowered memory footprint of LittleD is thanks in large part to its novel query parser that was written with the specific intent to lower energy cost, coupled with a query translation system that can build query execution

plans directly from the query itself without performing memory-consuming intermediate steps like building parse trees and logical query trees.

2.7.5 IonDB

Similar to LittleD, IonDB [FHD+15] is a database system that was designed for use on embedded systems. However, IonDB implements a key-value storage system instead of an SQL-compliant database like the one in LittleD. Key-value databases (also called NoSQL databases) do not implement the relational logic present in SQL databases, but are more memory efficient and streamlined as a result. Values stored in a NoSQL database are accessed using the key that was provided to the database at the time of insertion. Developers who do not need the additional features provided by a relational database may instead choose to use a NoSQL database for their simplicity and lower hardware overhead. In the context of an embedded system running on battery power, the lower overhead of a NoSQL may reduce energy consumption and thus extend the operating life of the device.

IonDB implements four different data structure options under a unified database API. These data structures are:

- Skip List
- Open Address Hash Map
- Flat File: This option does not use a data structure and appends data in no particular order onto a flat file structure (for scenarios where data values exceed or almost exceed the amount of available memory).
- File Based Open Address Hash Map

IonDB documentation provides a list of relative performance metrics used to compare its included indexes (listed in Table 2.4), however it does not include any performance comparisons to indexing structures that are unsupported by IonDB.

Data Structure	Memory Utilization	Runtime
Skip List	Fair	Excellent
Open Address Hash Map	Excellent	Good
Flat File	Good	Fair
File Based Open Address Hash Map	Excellent	Fair

Table 2.4: Relative Performance of the Data Structures Used in IonDB

Additional details of these indexing methods can be found in Section 2.8.

2.7.6 SBITS

In pursuit of faster response times, reduced network transmissions and lower energy usage on IoT sensor networks, edge-side querying and data analysis has been gaining in popularity. The Sequential Bitmap Indexing for Time-series Technique (SBITS) [FOL21] is an index structure optimized for the embedded devices commonly used as edge nodes in sensor networks. SBITS prioritizes the efficiency of its index structure to ensure it can run on embedded devices with limited memory and processing resources. Compared to general database servers, the queries performed on an embedded database are very well defined, and are known ahead of time. Outlier detection (determining if a value falls outside a specified range), and aggregation operations (average, max, min, etc) are the two types of queries that are typically fulfilled by embedded sensor nodes. This knowledge can be exploited to implement an indexing structure optimized for these queries. In SBITS, each data page includes a header that describes pertinent aggregate information about the records within the page. The items stored within the header include:

- The largest and smallest timestamp values
- The minimum and maximum values
- The summed total of all values within the page
- A bitmap describing the general distribution of values

The page-level aggregate values are used to fulfill aggregation queries for larger ranges without needing to read each stored value at query execution time. The bitmap is capable of informing the existence of records that fall within a configurable value range. If the bitmap is configured beforehand to identify values that fall outside of an expected range, outlier detection queries can use the bitmap to quickly process data pages without needing to read each individual data value. When querying by timestamp, the ordered nature of the dataset enables page retrieval to be calculated in $O(1)$ time. This is made possible by the use of consistent record sizes such that it is known how many records fit into a data page. By applying a linear regression, the location of any given timestamp can be calculated without conducting any reads. When analyzing memory consumption and querying performance, SBITS is able to match the memory footprint of the Antelope index, while

adding support for value-based queries. SBITS can also lookup timestamp indexes in constant time, whereas Antelope uses a binary search method which has a complexity of $O(log_2 N)$.

2.8 Conventional Indexes on Embedded Databases

For many database operations such as joins and other operations using comparisons, indexes play a major role in the optimization and performance of query execution. The TinyDB and Cougar architectures are purely acquisitional, meaning that only the most recent data tuple is gathered at the time of query execution. This architecture has no need for an index since sensor nodes only hold a single tuple at a time. However, with architectures that store relations filled with a large number of tuples, the existence of an ordered index on an unordered list of data tuples can speed up performance drastically. We can see this in the experimental benchmark results that were performed in the work for LittleD [DL14], where Antelope outperforms LittleD due to its better indexing support, but exhibits equivalent performance to LittleD when indexes are removed for both Antelope and LittleD. The various indexes applied by Antelope, LittleD, IonDB, and SBITS will be explored in this section, along with the rationale behind their usage.

2.8.1 Linear Hashing

The linear hashing data structure [FOM+19] has the desirable attribute of being able to expand one data "bucket" at a time, compared with extendible hashing which must double its capacity at every expansion. This is accomplished by splitting data buckets individually, so only one new bucket is needed to store the split contents (assuming no overflow buckets). In [FOM+19], two ways of optimizing the linear hashing data structure for use on embedded devices have been evaluated against the base implementation.

- The Bucket Map optimization removes the need for a secondary file to store overflow buckets by always appending overflow buckets to the end of the file. An in-memory array is then used to map these overflow buckets to their associated top-level bucket.

- The Serial Writing optimization expands upon the Bucket Map implementation by also writing changed buckets to the end of the file in addition to the overflow buckets. A similar in-memory array to keep track of the top-level buckets is used.

Surprisingly, the flash-aware optimizations did not result in a consistent performance improvement over the base implementation. The increased insert performance of the Bucket Map optimization led to a trade-off in other areas, and the Serial Writing optimization resulted in large file sizes which led to increased random writes (costly on the FAT16 file system used on the embedded test platform).

2.8.2 B-Tree Index

B-Trees are a commonly used indexing structure in high performance database applications, however they have large memory footprints that make them impractical for used in an embedded system. An adaptation of the B-Tree data structure to optimize memory usage [OFL21] aims to address this issue while also maintaining indexing performance. This adaptation also recognizes the possibility of unexpected power cycling that often occurs in embedded environments, and thus stores data in persistent storage rather than in RAM. This implementation of the B-Tree only requires two buffers, one for reading pages, and the other for writing pages. The entire B-Tree implementation can occupy less than 1.5 KB of memory and requires only around 10 KB of storage, making it practical for use in embedded systems. Additional memory buffers can also be used in this implementation for increased performance, though it is not necessary to do so.

2.8.3 MaxHeap Index

MaxHeap is a general purpose index developed for use on SD cards and NOR flash. Previous index types typically rely on balanced tree data structures, which must intermittently perform maintenance rebalancing operations. This operation requires rearranging elements on the tree structure, which is unsuitable for the write restrictions of NAND flash. Consequently, these structures use log structures which can have large memory footprints that are hard to fit inside the limited RAM found on embedded devices. MaxHeap aims to address this design consideration by implementing a data structure designed for fewer writes. The binary maximum heap structure fits these specifications, and it is used in the MaxHeap index to naturally map a dynamically expanding dataset. The MaxHeap structure uses two files for every indexed attribute. One file, called the bucket set container, contains the key-value pairs that form the index. The other file, called the heap descriptor, contains nodes that describe the range of keys stored inside the nodes inside the bucket set container. The MaxHeap structure is populated

from the top down, with new elements being inserted into the bucket that has the narrowest range around the key to be inserted. Whenever a bucket is filled to capacity, it is split into two separate buckets that are inserted as children to the original bucket that reached capacity. This scheme can behave problematically when keys are presented in an ascending or descending order, resulting in an unbalanced tree. This issue is addressed by using a hashing function to generate the binary heap keys, but storing the un-hashed key in the actual bucket. MaxHeap indexes accomplish their goal of being a memory efficient indexing scheme that works well for general datasets and can be operated within the RAM constraints of an embedded device.

2.8.4 Hash Index

Antelope and IonDB both implement hash indexes, which store a hash table in RAM to quickly look up commonly accessed attributes that have relatively low row counts. This approach is extremely fast, but is also very memory hungry so it must be used sparingly and only in circumstances where a few keys are accessed very frequently. The exact limit on the number of keys that this index can successfully accommodate varies depending on hardware and design limitations, so Antelope designates a default maximum number of 100 tuples for its hash index. IonDB's "Open Address Hash Map" takes a different approach to hashing that stores buckets of data in flash rather than keeping an dense table of tuple pointers in memory. This data structure is discussed in detail in Section 2.3.3. In summary, a deterministic hashing algorithm is used to assign incoming key-value pairs into buckets, which allows other buckets to be omitted when querying using a search key. IonDB can also use "File Based Open Address Hash Map" which is similar to the regular Open Address Hash Map implementation, except the hash map structure is stored in storage rather than memory.

2.8.5 Inline Index

While the MaxHeap and hash indexes implemented in Antelope cover a wide range of usage scenarios, they assume that the distribution of keys to be inserted will be in random order. This is useful when indexing attributes that follow a normal distribution of values such as temperature sensor data, however with embedded sensor networks the data collected tend to be time series datasets resulting from the repeated sampling of a sensor throughout a reporting period. Time series datasets have the unique characteristic of using timestamps as index keys, which are generated in ascending order. A

set of constantly increasing keys (such as the ones in a time series index) can be exploited using the inline indexes found in Antelope and SBITS, which do not use any data structure. Instead, the inline index uses the ordered nature of the indexed keys to perform a binary search over the entire range of keys when conducting a search. This results in a O(1) space overhead while maintaining a O(logN) time complexity for search operations [TD11]. SBITS used a single linear approximation of the time series values to predict the location of a given timestamp.

2.8.6 Skip List Index

IonDB includes the option to use skip list indexes in scenarios where the average access time for all tuples is important. Additionally, skip list indexes do not require in-place updates like balanced tree data structures. Implementations of tree structures on devices using flash storage also incorporate separate data structures (usually a log structure [RO92]). This structure is used to delay or batch writes together for the purpose of reducing expensive in-place updates. These auxiliary data structures also take up valuable memory within a memory-constrained embedded device, which leaves less memory available for the tree structure itself. Skip lists do not require auxiliary data structures, as they can be implemented as append-only fashion in scenarios where inserted data has keys that are monotonically increasing. The skip list data structure is discussed in detail in Section 2.3.2. In summary, skip lists use probabilistic models that link to the most commonly accessed sections within a list, "skipping" over the less-frequently accessed elements.

2.9 Time Series Indexes

When retrieving the data associated with a certain timestamp, the location of the page containing the correct tuple must be determined.

A naive approach of conducting a binary search through all the data pages will accomplish this, however this tends to be an expensive search operation as each IO requires reading a data page and checking its contents, which takes a considerable amount of time, adding to the overall latency of the data retrieval operation. To minimize the number of required IO operations needed to find the correct data page, an indexing structure can be applied to the timestamps of the time series database.

One of the simplest database indexes is the linear approximation. This is a simple linear regression that is fitted over the timestamps of the dataset.

Despite its simplicity, this index has seen success in applications where the time series records are equally spaced apart, such as in embedded sensor networks configured to generate records with consistent sampling rates. The SBITS [FOL21] index for embedded sensor networks applies a linear index over its collected data, and the simplicity of the linear approximation index allows it to run efficiently on an embedded device.

More conventional indexes are used in applications where the timestamps are not guaranteed to be evenly spaced apart. B-Tree indexes are a well studied topic, and are commonly used in database management software. The LittleTable [RWW+17] time series database makes good use of B-Tree indexes, allowing it to group relevant data together so that it can be read sequentially to improve performance. B-Tree indexes have a larger memory footprint than indexes based on a linear regression, making them more difficult to implement on low power devices. B-Tree implementation for embedded devices [OFL21] has good performance but is not optimized for sorted timestamp keys.

Learned indexes attempt to retain the flexibility of conventional indexes while reducing the memory footprint so that they can be used on a wider variety of devices. In recent years, learned indexes have seen considerable research, resulting in a number of different approaches which will be explored in the following section.

2.10 Learned Indexes

Conventional database indexes typically utilize data structures that have been optimized for a large variety of dataset types to ensure database performance remains adequate in a wide range of usage scenarios. Contrarily, some other database indexes use data structures that are optimized for a specific type of dataset. These specialized database indexes outperform their more general-use counterparts in specific use cases, but may have other use cases in which they perform much worse or are wholly inapplicable. Learned indexes seek to combine the versatility of general-use database indexes with the performance of specialized indexes by devising an indexing scheme that dynamically optimizes for each individual dataset, rather than a type of dataset or a wide range of datasets. Machine Learning (ML) techniques are commonly used in learned indexes, as they can train on samples of datasets to learn the patterns and trends within a specific dataset. Database indexes can be reformulated into common ML problems which have already seen extensive research and development, such that existing ML techniques can be

applied. For example, B-Tree indexes can be reformulated as a regression tree problem, as the main point of such indexes is to map keys to a position within a larger range of values [KBC+17]. Figure 2.12 shows the structure of a typical regression tree with ML models forming the nodes. ML models are also able to train on the type of queries that are most commonly executed in deployment. This allows for the model to further optimize for a specific usage scenario by optimizing the indexing structure to favour the parts of the dataset that are most frequently accessed in addition to the overall structure of the dataset itself. The resulting generated model takes the place of the data structure used in conventional database indexes.

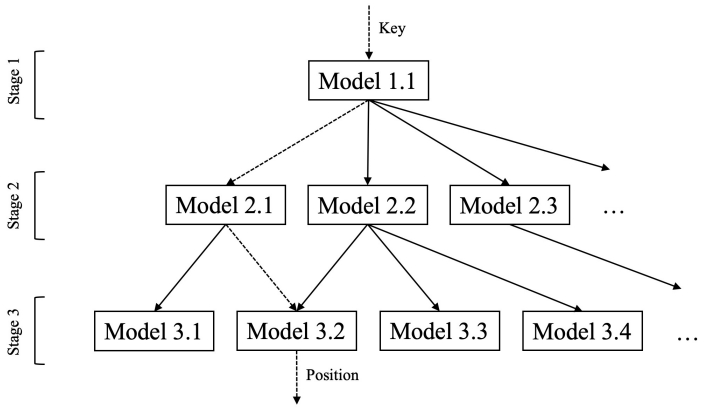

Figure 2.12: Learned Indexes

Mathematical models can also be used in learned indexes. These models can be built with guarantees of error and transparency of functionality that are difficult to achieve with ML models. Learned indexes can be built from the bottom up, or from the top down. Models created from the top down are trained to capture the overall trend of a dataset as the top layer, then lower layers are used to narrow down the search range and reduce error until the required specificity is achieved. This has the benefit of being able to closely optimize for datasets, but tend to be inflexible to changing data trends and may need periodic retraining to maintain performance. Models created from the bottom up first fit the bottom layer to the points within the dataset to within a tuneable error bound. Additional upper layers are then built on top of the bottom layer to optimize querying performance when searching the bottom layer. Certain bottom up approaches have the capability to dynamically fit their models to suit new data points as they are inserted

into the database, resulting in a model that is more flexible to changing data trends. In this work, the PGM and RadixSpline indexes will be benchmarked and compared to gather performance data using real-world datasets.

2.10.1 RMI

The creation of the Recursive Model Index (RMI) [KBC+17] was motivated by the lacklustre performance of implementing a learned index using existing tools in Tensorflow and Python. These tools were designed to run large, complex models and have significant invocation overhead when running the relatively simple models used in learned indexes. Additionally, naive learned indexes composed of ReLU nodes are good at generalizing the overall trend of an index distribution, but struggle when asked to produce precise results with a narrow error range.

RMI addresses these limitations by incorporating a streamlined system for executing learned models without the overhead associated with Tensorflow, Python, and other existing tools. Dubbed the Learning Index Framework (LIF), this framework is capable of extracting the weights of a trained Tensorflow model, and execute it efficiently in C++ without invoking an actual instance of Tensorflow. Using this framework, a recursive hierarchy of learned models is built, with each model selecting the next model to be used on the next level down the hierarchy of models. The general structure of RMI is shown in Figure 2.12. This approach recursively appoints a segment of indexes that a single model is responsible for. Compared to using a larger, more complex model to predict indexes with the same granularity and accuracy, a hierarchical collection of models is able to decouple its overall size and complexity from the execution cost. This is partially attributed to the fact that only a few select models are evaluated with any given index query, whereas a monolithic model must be evaluated in its entirety to fulfill an index query. Additionally, the upper level models of the hierarchy are still able to learn the overall shape of the distribution of indexes. This exploits the ability of neural networks to fit general trends in the data and still retain accuracy by leaving the precise predictions to other models that are trained separately.

The RMI model is created using a recursive method from the top-down, starting with a model trained on the complete dataset. Using the predictions generated from this root-level model, the models to be used on the next level down are chosen and the keys that this lower-level model is responsible for are added to a list. This list is then used to perform the actual training on the lower-level model. This process is repeated recursively for a specified

number of models per level and total number of levels. The errors for the models on the last levels are also stored so that the errors can be compared against the error threshold set by the user. In the case of an exceptionally hard-to-learn data distribution where a model's error exceeds the threshold, that model is replaced with a B-tree.

This recursive structure of models are able to significantly outperform conventional B-tree indexes on a dataset composed of 200M log entries for requests to a university website while being a fraction of the size of the B-tree. While this is impressive, this indexing structure was developed with the datacenter in mind, requiring the sequential computation of multiple neural networks with numerous hidden layers each. For systems with heavily constrained hardware resources, other options are also available for consideration.

2.10.2 PLA

The PLA (Piece-wise Linear Approximation) functions [EEC+09a] are a set of two filtering techniques that are applicable to time series indexing. These filters aim to compress the space requirements needed to store a time series dataset within a given error bound, while minimizing memory usage and maximizing the compression ratio. The two filters introduced are named the Swing filter and the Slide filter. They are both approaches that utilize linear functions to compress an input stream of data. This is similar to the segmented linear regression problem, where linear functions of least squared error are fitted to sub-divided segments of a larger data set. Ways to determine the best segment sub-partitions have been the subject of previous research in the past, where it has been solved using dynamic programming techniques [Fed75]. However, unlike the segmented linear regression problem, an important distinction of PLAs is that it does not minimize the error of the linear functions. PLAs fit the underlying data points with a guaranteed error bounds set by the user, and thus allow for more flexibility for use as an index.

The Swing filter implementation proposed by the authors uses a series of continuous linear line segments that approximate the input data. The first element to be inserted into the swing filter forms the origin point of the first linear line segment. After the first point is inserted, a pair of bounding slope lines are created with their origins at the inserted point. One line represents the maximum slope that the current line segment can possess while still keeping the guaranteed error bounds, and the other line represents the minimum slope. Subsequent points entered into the swing filter are

evaluated to see if they land within the region bounded by these lines. If the new point lands within the region, then this point is not recorded as it can be represented by the current linear segment. Instead of recording the segment, the slope lines "swing" up or down to reflect the new maximum and minimum slopes of the current line segment (that now incorporates the newly inserted point). If the new point falls outside of the region, the current line segment ends. An optimization process determines the best slope that minimizes the mean square error of all the points covered by the current line segment, then ensures that it falls in between the upper and lower bounding slope lines. This process uses a procedural summation process that can be generated iteratively and does not require the buffering of any points. The generated slope is then extrapolated to form an optimized point to be recorded. It is also used as the starting point for the creation of a new line segment to cover future insertions.

Similar to the Swing filter, the Slide filter maintains a set of upper and lower hyperplanes. However unlike the Swing filter, the Slide filter is not guaranteed to generate a set of continuous linear segments. This means that the starting point of a given line and the ending point of the previous line are not guaranteed to be the same point. The Slide filter describes a method on how to choose upper and lower bounding hyperplanes. In summary, given a set of points to be covered by a single line segment $(t_k, x_k), (t_{k+1}, x_{k+1}), (t_{k+2}, x_{k+2}), (t_{k+3}, x_{k+3}) \dots (t_n, x_n)$ the lines that pass through (t_k, x_k), and (t_{k+1}, x_{k+1}) for all points such that $i >= 1$, and all points are within ϵ in the x-axis should be considered as candidate bounding upper and lower bounding lines. The candidate line with the lowest slope should be chosen as the bounding line for the upper bound, and the line with the highest slope should be chosen as the bounding line for the lower bound. Another process further optimizes bounding line selection by proving that only the points belonging to the outer hull of the points encountered need to be evaluated for the bounding line candidacy, while the other points can be safely added without selecting new upper and lower bounding lines. Furthermore, the Slide filter also attempts to create continuous lines by offsetting the starting point of a new line. This is only possible if an intersect between the current line starting at (t_k, x_k) and the previous line ending at (t_{k-1}, x_{k-1}) occurs between t_k and t_{k-1}. The slide filter algorithm uses these upper and lower bounds to calculate an optimal series of lines using the same slope optimization method used in the Swing filter. These line segments are then recorded in place of points.

2.10.3 PGM

The PGM index [FV20] operates in a similar manner to the PLA Slide filter, as it also dynamically creates a series of linear models which gives an approximate page location within a configurable guaranteed error margin. However, instead of maintaining the minimum and maximum slope lines for each linear segment, the PGM index forms linear line approximations using a greedy method which attempts to fit a bounding box with a height of 2ϵ around the incoming data points. This bounding box can be rotated to orient itself along any slope to fit points. When PGM encounters a new point that does not fit within the bounding box containing the previously encountered points, the line representing the previous bounding box is added to the index as a PGM linear segment, and a new bounding box is created. This process also forgoes the optimization process described in the Slide filter, resulting in a less optimal but also less resource intensive indexing scheme that is better suited for resource constrained embedded devices. PGM linear segments are represented by three components, a slope (float), a key (int), and a x-intercept (int). The float and the x intercept are used to define the line itself, and the key is used to specify the starting range when the linear line segment should be used. To query this index, we conduct a binary search on the keys of the PGM linear segments to find the correct linear segment that services the range of timestamps being requested. After the correct linear segment is found, the slope and x-intercept of the line segment are combined with the queried timestamp to generate a page number that is guaranteed to be within ϵ of the real page number. A PGM model fitted to a time series dataset is shown in Figure 2.13, note the disjointed line segments generated by the PGM model. Table 2.5 shows the approximate values of the linear segments generated by PGM, and the calculations on Figure 2.13 uses this table to compute the approximate page location of the record recorded with a timestamp of 610k. The slope and intercept of the PGM entry with the next lowest key (compared to the search key) is used to compute the line equation which generates an approximate page location of the search key.

Key	Intercept	Slope
588k	0	$6.63 * 10^{-4}$
600k	9	$9.87 * 10^{-4}$
606k	16	$1.37 * 10^{-3}$
620k	37	$1.83 * 10^{-3}$

Table 2.5: PGM Index Points

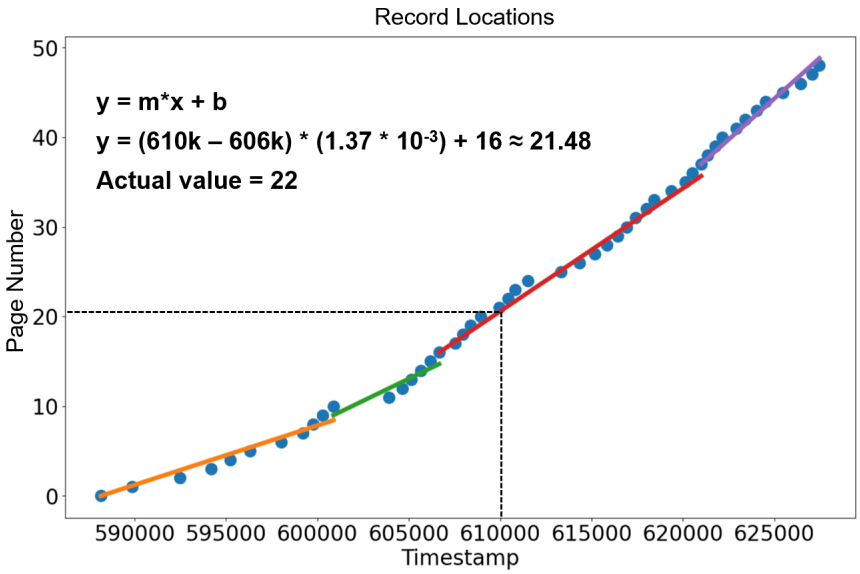

Figure 2.13: PGM Query

2.10.4 Spline

In the original work for the Radix Spline indexing model [KMvR+20], the argument was made that an efficient build process is sufficient to replace the functionality for supporting insert operations. This efficiency is very advantageous in an embedded environment such as a distributed sensor network, and the lack of insert support can be averted since the sensor network will only append the latest readings into a time-series database. Like the PGM model, the Spline model also creates a series of linear segments that approximate the page locations for a certain timestamp with a configurable guaranteed error. Unlike the PGM, spline does not create independent lines that are fit over a defined range of timestamps. Rather, spline forms linear line segments by connecting points to one another similar to the contiguous linear segments generated by the PLA Swing filter. Unlike the Swing filter, Spline does not keep track of the minimum and maximum bounding slope lines. Instead, Spline points are chosen using a greedy method that evaluates whether a new point would violate the error guarantees defined by the user. Starting with a single origin point, a line is drawn from the origin point to each new point as they are entered indexed by the spline model. If the new point violates the error bounds, a spline linear segment is created that connects the origin point with the last point which did not violate the

error constraint. The last point then also becomes the new origin point, and a new spline line segment is started. Much like PGM, the Spline process also forgoes the optimization process described in PLA which attempts to minimize the mean square error of the generated line, but also results in a simpler algorithm that is well suited to run on hardware constrained embedded devices. A Spline model fitted to a time series dataset is shown in Figure 2.14, note the continuous line segments generated by the Spline model. Table 2.6 shows the approximate values of the linear segments generated by the Spline index, and the calculations on Figure 2.14 uses this table to compute the approximate page location of the record recorded with a timestamp of 610k. The two bounding spline points of the search key are located in the spline table, and the slope between them are calculated. This slope is then multiplied by the offset of the search key from the lower spline point, and the min value is added as the intercept to generate the approximate page location of the search key.

Key	Min Value
599k	7
601k	10
605k	12
611k	24
616k	29
621k	39
627k	48

Table 2.6: Spline Points

In the original work, a Radix indexing scheme was applied on top of the Spline model. This additional index is used to speed up access times on very large datasets that generate a large number of spline points. In the use case benchmarked in this work, the speed increases were minimal and not worth the additional memory overhead required to store the radix index. The querying process for the spline index is similar to that of the PGM index. First, a binary search is conducted on the keys of the spline line segments to find the correct segment that services the range of timestamps that the queried timestamp resides within. The slope between the current spline point and the next spline point is computed. This slope is then used as a linear interpolation to return a page number guaranteed to be within ϵ of the true page number which contains the data corresponding with the queried timestamp.

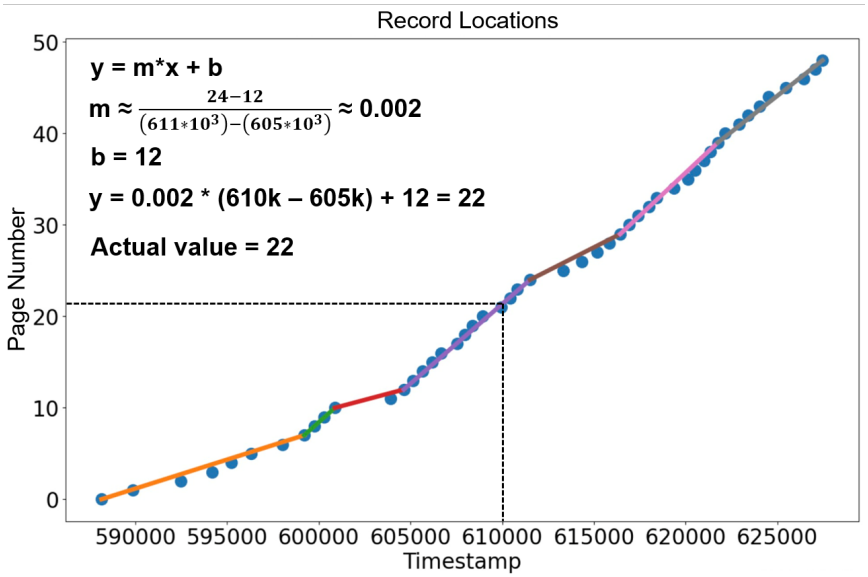

Figure 2.14: Spline Structure

2.10.5 Comparison between Spline and PGM

Both Spline and PGM create a series of linear line segments that provide an approximate page number that contains the data for a given input key. The advantage of PGM lies in the fact that it can create completely independent line segments to represent the underlying data. The Spline model, by comparison, must create lines that are formed by drawing lines between existing data points of the data set. This generates less optimal linear line segments compared to the lines generated by PGM, but in return, Spline is able to use less data to store each individual line segment as it only needs to store a key and a page number. By comparison, PGM must store a key and page number, plus an additional slope variable for each line segment. In the later experimental sections, we will explore the real world performance of both indexing methods, along with the performance of binary search and a simple linear model across the entire dataset.

When applied to time series datasets, both indexes are able to generate a series of linear approximations fitted to the underlying data. A graphical comparison between the two models is shown in Figure 2.15. Note the disjointed nature of the PGM approximation, and the continuous nature of the Spline approximation. This is a result of the PGM model being capable of generating completely independent lines for each linear segment, whereas the

linear segments generated by the Spline model must originate and terminate on existing data points.

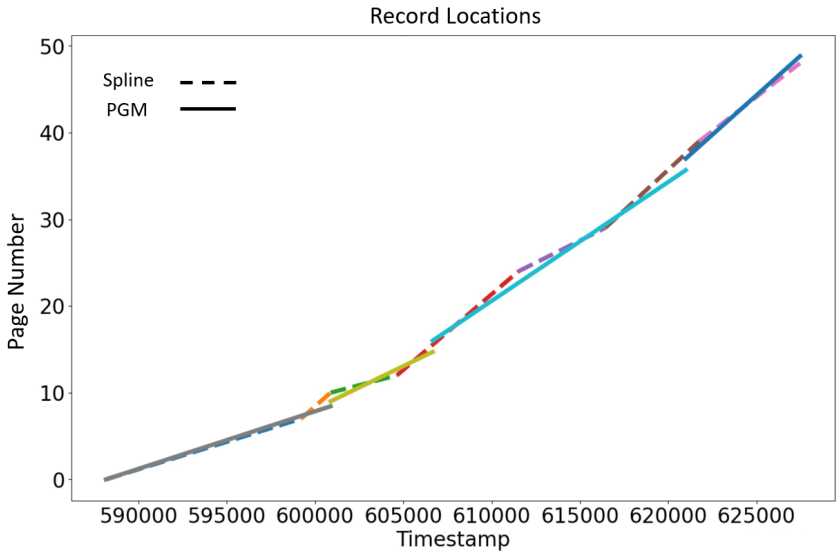

Figure 2.15: Spline and PGM Models Fitted Over Time Series Data

Chapter 3

Methodology

To create an index structure for use in embedded sensor nodes, it is important to understand the data operations that sensor nodes perform while they are in use. Embedded sensor nodes collect sensor values and store them for future retrieval and processing. Attached sensors are periodically polled to generate new readings, which are stored in sequential order on a storage device. This generates a dataset of sensor records that are naturally ordered by the timestamp at which they were taken. These sensor records are iteratively appended to storage as they are taken and are never modified, resulting in an append-only database. An index applied on this data will always access an ordered dataset, but will need to support iterative appends as new sensor readings are taken at subsequent sampling intervals.

A learned index model adapted for use in embedded sensor nodes must 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.

An ϵ value of 1 was chosen for both the PGM and Spline learned indexes. This error bound represents a balance between memory usage and accuracy, where a ϵ of 0 is very strict and generates a large number of index entries. The size of the data sets used to generate experimental results (shown in Table 5.2) did not necessitate any further reductions in memory consumption. Further relaxations of the error bounds do translate into an additional reduction in memory consumption, but these reductions are not as large as the initial memory reduction from changing ϵ from 0 to 1 (as demonstrated using the SEA data set in Table 5.4). The radix table was omitted from the

RadixSpline index, as relatively small number of spline index points generated did not benefit from using an additional radix table to speed up lookup operations for spline points. These tuning decisions will be further explored in Section 5.

The indexing approach is based on SBITS [FOL21] 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.

This procedure was implemented with efficiency in mind. By buffering data records in memory until a full page can be written, we avoid needing to update the same data page on the storage multiple times for multiple data records, which helps to minimize the erase-before-write characteristics of flash storage. These data pages are then stored in the order which they are created, which is ordered by timestamp. This ordered set of pages are then inserted into the indexing algorithm, which uses the minimum timestamp of each page as the search bounds for queries returned by the index. Generated indexes often have less than 1 KB due to the efficiency of learned indexing structures. This is beneficial to scale down hardware requirements for deploying learned indexing structures on lower power embedded devices which may have power consumption and cost advantages.

SBITS has been demonstrated to outperform conventional B-tree and hashing indexes in time series data [OKFL22b], so comparisons with those data structures are not performed. Additionally, when inserting records in an ordered, ascending manner, the self-balancing characteristic of B-tree data structures means that it must perform frequent re-balancing operations to avoid becoming right-heavy. These re-balancing operations incur significant overhead compared with linear indexes and the Spline and PGM learned indexes being evaluated, since they are generated iteratively and do not require such operations. 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 piecewise linear approximations (PLA) [FV20]. 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'R81] and rediscovered by [EEC+09b]. 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 [EEC+09b]. The Swing Filter does not yield an optimal number of lines, but requires only $\mathcal{O}(1)$ time and memory to process a point. This is a benefit in the sensor use case, as the optimal Slide Filter requires $\mathcal{O}(h)$ 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).

3.2 RadixSpline

The RadixSpline is a learned index that approximates the CDF via a linear spline and a radix table storing spline points [KMvR+20]. 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 spline over the empirical CDF using the greedy algorithm proposed by [NM08]. 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

Algorithm 1: AppendPGMAdd(pgm, x)

```
L ← pgm.countLevels();
K ← x;
for i ← 0 to L − 1 do
    c ← pgm.levels[i].countPoints();
    pgm.levels[i].add(K); // add implements the Swing Filter
    if pgm.levels[i].countPoints() > c then
        K ← pgm.levels[i].getLastKey();
    else
        return;
    end
end
/* If the algorithm reached this step create a new level    */
pgm.levels[L] = newPGMLevel();
firstK ← pgm.levels[L-1].getFirstKey();
lastK ← pgm.levels[L-1].getLastKey();
pgm.levels[L].add(firstK);
pgm.levels[L].add(lastK);
return;
```

Algorithm 2: QueryAppendPGM(pgm, x)

```
if x < pgm.levels[0].getFirstKey() then
    return NotFound;
end
L ← pgm.countLevels();
m← 0; // index of the model at the i-th level
for i ← L − 1 to 1 do
    a ← pgm.levels[i].getSlope(m);
    b ← pgm.levels[i].getIntercept(m);
    pos ← ⌊a · x + b⌋;
    lo ← max{0, pos − ε − 1};
    hi ← min{pgm.levels[i].countKeys() − 1, pos + ε + 1};
    m ← smallest value of j such that x ≥ pgm.levels[i-1].getKey(j) and
      lo ≤ j ≤ hi;
end
a ← pgm.levels[0].getSlope(m);
b ← pgm.levels[0].getIntercept(m);
pos ← ⌊a · x + b⌋;
lo ← max{0, pos − ε − 1};
hi ← min{pos + ε + 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);
```

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 [NM08], 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 ← rs.spline.countPoints();
rs.spline.add(x); // Add implements GreedySpline
if rs.spline.countPoints() > c then
    K ← rs.spline.getLastKey(); RadixTableInsert(rs, K, c); // see
       Algorithm 4
    return;
end
return;
```

Algorithm 4: RadixTableInsert(rs, K, pos)

```
r ← rs.r; // radix prefix size
K ← K - rs.minKey;
nB ← mostSignificantBit(bitShiftRight(K, rs.shiftSize)); // number of
   bits required to fit on the table
Δ ← max(nB - r, 0); // difference between the available and
   required number of bits
if Δ > 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 ← rs.shiftSize + Δ;
    newTable ← allocateTable( 2^r );
    for i ← 0 to 2^r do
        j ← bitShiftRight(i, Δ);
        newTable[j] ← min(rs.table[i], newTable[j]);
    end
    for i ← 2^(r−Δ) to 2^r do
        newTable[i] ← INT_MAX;
    end
    rs.table ← newTable;
end
// Update entry of the radix table with the new pos
T ← bitShiftRight(K, rs.prefixSize);
rs.table[T] ← min(rs.table[T], pos);
return;
```

Chapter 4

Implementation

4.1 Radix Adaptation

As discussed, append support for the RadixSpline was implemented using the streaming GreedySpline [NM08]. To accommodate this feature, the Radix portion of RadixSpline was also modified to support appends. The Radix structure operates over a set of key-value pairs, mapping a specified number of leading bits from keys to the values associated with the keys. This results in a mapping structure where the value of the first encountered key with a certain bit prefix is mapped to all subsequently encountered keys that share the same bit prefix. Time series data is well suited for the Radix structure as it naturally maps timestamps to the bounding search index in an array of time series data. The approach for supporting appends on Radix was similar to the approach for GreedySpline. In the base Radix implementation, the first point inserted into the Radix structure is stored as the 'minKey' value that will be subtracted from future incoming values. Future incoming points will then have minKey deducted from their values, the resulting value will have its leading zeroes counted. This count is used to verify whether the number of significant bits of the incoming key will fit within the number bits that the Radix table was initialized with. The resulting value with the leading zeroes removed is called the 'prefix', which will then be checked to see if it will fit within the specified number of Radix bits and indexed accordingly if it is. If the prefix does contain more bits than the specified number of Radix bits, the least significant bits are discarded from the prefix (right -shifted) until the prefix fits inside the Radix index. The base implementation of the Radix table did not support continuous appends, and as such did not account for a variable number of significant bits from incoming keys. This variability changes the number of right-shifts required on incoming keys before they are indexed. To remedy this, the current number of right-shifts are stored in memory and compared against the right shifts required by future keys. If an incoming key has a higher number of significant bits, a refactoring process is executed on the existing Radix table. This process shifts all existing values in the Radix table by the difference in significant

bits of the incoming key and the stored right-shift value. After this process, all keys which share significant bits within the Radix table are discarded, only leaving the first key-value mapping that matches the updated prefix. This process repeats each time a new key is inserted into the Radix structure with a higher number of significant bits, allowing for a continual insertion of increasing keys. Figure 4.1 illustrates the refactoring process that occurs when a new data entry (8 in this example) is inserted into a fully populated radix table. Note the shifted prefixes, and how the radix becomes a sparse index (having been a dense index before). A dotted line is used to indicate that the new maximum value that can be stored using the refactored radix table is 15, while the previous maximum value was 7.

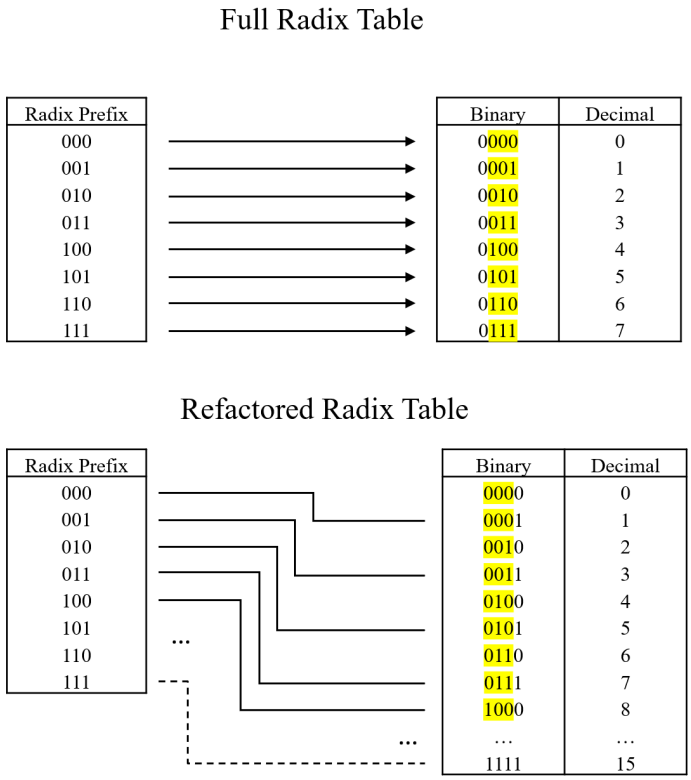

Figure 4.1: Radix Table Expansion

4.2 Memory

The PGM and Radix Spline index structures are both stored in a dynamically allocated block of contiguous memory. The location estimates generated by the indexing mechanisms are then applied as byte offsets to the storage pointer, where the corresponding index value can be retrieved.

4.3 SD Cards

SD cards are a popular choice for embedded devices due to their low cost and compatibility with the SPI data communication interface. SD cards consist of a NAND package, and a controller chip to handle FTL operations within the chip that converts block addresses to physical flash addresses. SD cards can be configured to operate in SPI mode, which eliminates the need for an external SD controller to interface with the storage device. In this mode, SD cards operate like a standard block device, using standardized SPI commands for reading and writing data. In the sensor hardware test platform, the test datasets are loaded from an SD card to be inserted into the on-device database where the performance of the PGM and Spline indexes are evaluated.

4.4 Dataflash

The sensor hardware platform used to gather experimental results records all inserted data points onto dataflash storage[1]. Dataflash storage consists of NOR memory instead of the NAND memory found in conventional flash storage. NOR memory differs from NAND in its structure, which allows for each data page to be individually cleared whereas NAND flash must clear an entire block (consisting of multiple pages) at once [FPL16]. The data inserted into storage in testing is append-only, and inserted data is immutable. This storage type allows for records to be inserted one page at a time as they are taken, whereas NAND implementations must buffer an entire flash block at once before writing to storage to avoid incurring the overwrite penalty associated with in-place updates. Like SD cards, Dataflash also communicates with the sensor device using the SPI protocol.

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

4.5 SPI Protocol

Files stored on SD cards attached to the embedded device accessed via Serial Peripheral Interface (SPI) are managed by the SdFat library. The Dataflash storage are also attached using the SPI protocol.

4.6 FAT File System

The FAT file system is named after the File Allocation Tables it uses to keep track of data clusters on a disk. It is a popular choice for embedded applications due to its simplicity, maturity, and cost-effectiveness.

Compared with other file systems like NTFS or HFS, FAT has a simpler design that is easy to implement, has a smaller memory footprint, and requires less processing power. These characteristics make FAT very well suited for low power embedded devices with limited hardware resources.

The FAT file system is also supported by most modern operating systems, owing to its widespread usage as the default file system of the MS-DOS family of operating systems. This wide support-base makes inter-compatibility between different operating systems and devices much more convenient.

FAT is also open-source and royalty free, meaning that developers looking to implement the FAT file system do not have to pay licensing fees to a governing body. This makes FAT a cost effective solution for embedded devices that are typically deployed in large fleets.

The FAT file system was chosen to facilitate cross compatibility and easy transfer of datasets onto the hardware test platform when conducting experiments.

4.7 Data Transfer From SD Cards

The full process for reading the contents of a file from an SD card is as follows, all read operations are performed using the standardized SPI commands.

1. The boot sector of the SD card is read to find the location of the File Allocation Table (FAT).

2. The root file directory is read using the FAT table to find all the clusters that the root file occupies, then reading those clusters.

3. The root file directory contains information (including the location of their first clusters) on all the files and sub file directories stored in the root directory.

4. If the file of interest is not located in the root directory, sub file directories are traversed recursively until the correct sub directory is reached.

5. The file directory is read to find the starting cluster of the file of interest, and the FAT table is again used to locate all the clusters pertaining to the file being read.

6. Clusters are read one at a time into the host device, since limited resources are available and it is not advisable to read the entirety of a file into memory.

These operations are handled by the SdFat library. The process for writing a file onto the SD card using SPI is similar to the process for reading a file, the main difference being that the SPI command for write operations are used instead of read operations when the correct file is located.

4.8 Data Transfer to Dataflash using SPI

No file system is used on Dataflash to maintain simplicity and reduce overhead on the embedded device. Drivers for the Dataflash storage were created as part of the IonDB project [FHD+15] which had the capability to use the same type of NOR flash storage. The Dataflash drivers include functions for:

- Initializing the flash structure
- Erasing the entire flash chip
- Reading bytes from a flash data page
- Writing bytes into a flash data page

Since only a single file was used to store records, a file system is not necessary on the Dataflash storage. Compared to FAT, this streamlined raw storage system is easy to implement and reduces overall overhead.

4.9 Build Environment

All code is implemented in C due to its low abstraction, allowing direct access to pointers for manipulating data structures. C is also widely supported by the majority of major hardware platforms, enabling portability to systems manufactured by different vendors.

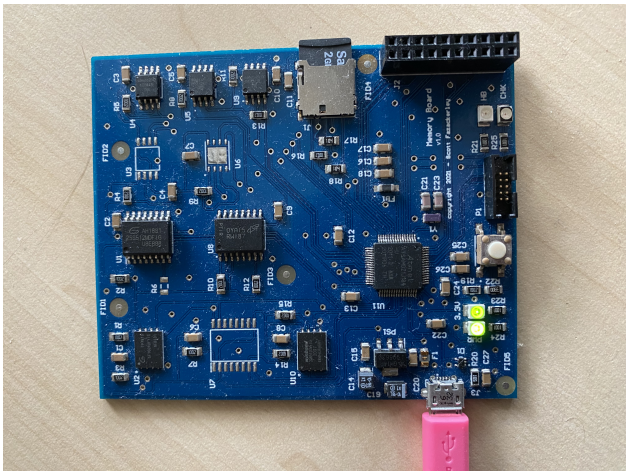

Figure 4.2: Custom Development Board With Integrated NOR Dataflash

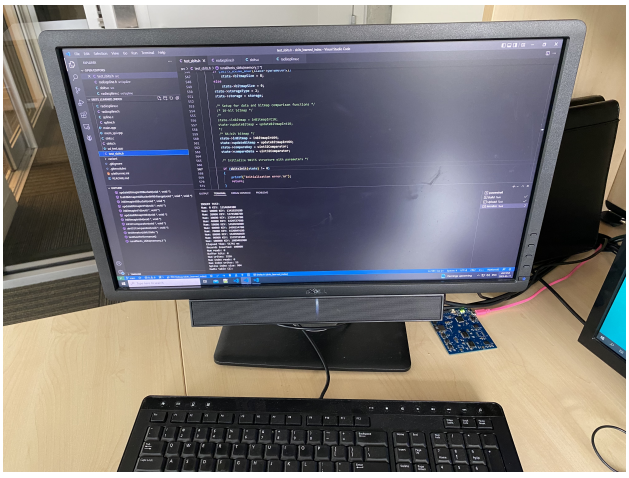

Figure 4.3: Desktop Environment of Platform.io Extension on Visual Studio Code

Tests were evaluated on a SAMD21 processor and a development board that has been modified to include onboard NOR dataflash storage (shown in Figure 4.2). Compilers and build setup for this platform is handled by Platform.io, which has a plugin for Visual Studio Code (pictured in Figure 4.3). It includes a library of compilers and needed build scripts for a variety of embedded platforms (Atmel SAM, Raspberry Pi, Espressif32, etc). In the case for the SAMD21 processor, the ARM GCC compiler is used.

Chapter 5

Results

We choose to evaluate querying performance using the Spline and PGM learned indexes as well as the linear index implemented in SBITS. The experiments measure four core metrics: the query throughput, the number of IOs 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.

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[2] 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 5.1.

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.

[2] `https://www.dialog-semiconductor.com/products/memory/dataflash-spi-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 IOs 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 5.2. 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 [ZYLK+05] and have been used in several further experimental comparisons [FOL21, OKFL22a]. We also present the eCDF for the datasets in Figure 5.1. From the eCDF graphs, the phone and watch datasets are of particular interest since they have the most variable sampling rates out of all the data sets used. Considering the ability of learned indexes to better fit an underlying eCDF compared with conventional indexes, experimental results generated using these data sets should be closely examined.

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

Table 5.1: Hardware Performance Characteristics

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	[ZGD+17]
ethylene	4,085,589	100,000	ethylene concentration	[FSHM15]
phone	18,354	18,000	smartphone X/Y/Z magnetic field	[BCRP16]
watch	2,865,713	100,000	smartwatch X/Y/Z gyroscope	[SBB+15]

Table 5.2: Experimental Data Sets

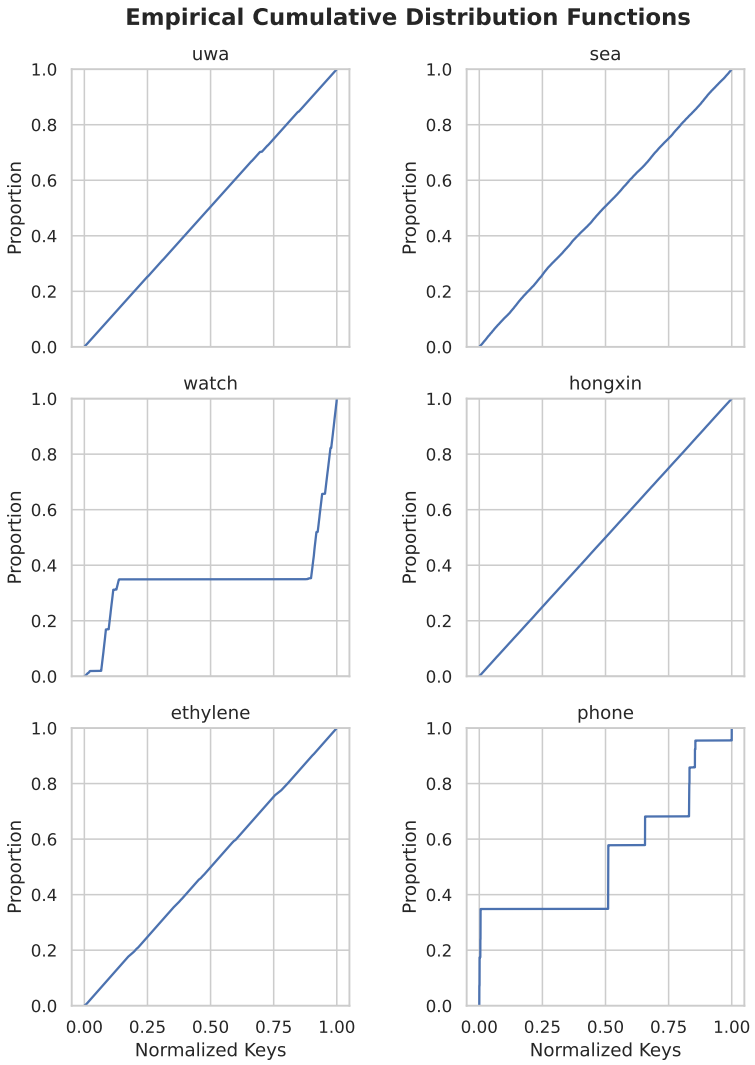

Figure 5.1: eCDFs for the experimental datasets.

5.1 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 IO operations performed.

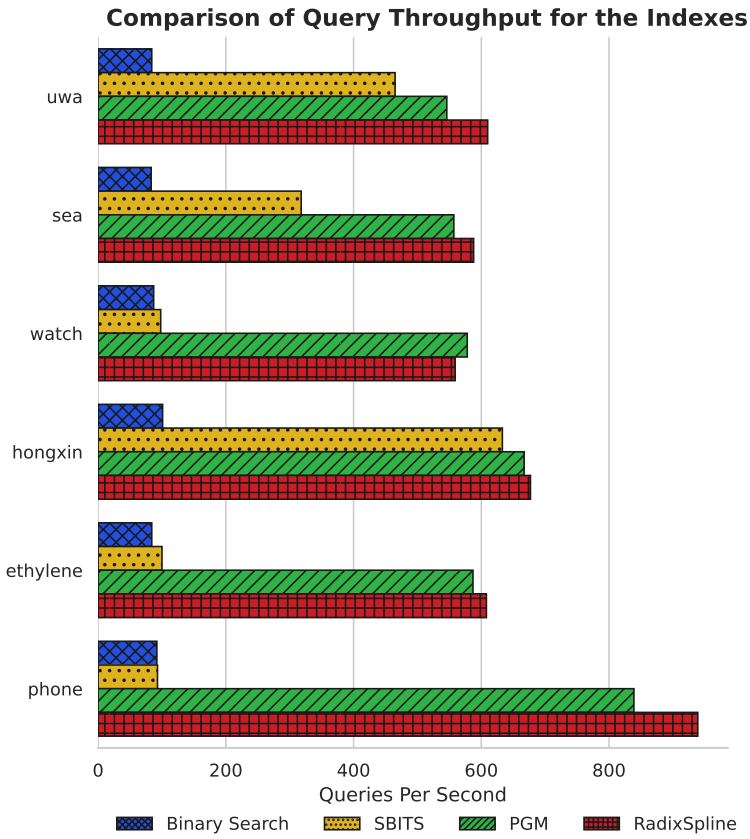

Figure 5.2: Average query throughput among the indexes for each dataset in the benchmark. Higher rates indicate better results. The RadixSpline and PGM consistently outperform SBITS and binary search.

The query throughput results are in Figure 5.2. 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. For the smart watch and smartphone devices that generated the **watch** and **phone** data sets, this increase in querying speeds will help enable more complex applications to run on these devices to efficiently process larger amounts of data. Additionally, this enables event driven and threshold driven data sampling, which has power consumption benefits as the device can remain in a low power state when it is not taking sensor readings. This is in contrast to fixed sampling, where the device must periodically take sensor readings at specified times regardless if the gathered data is meaningful. Learned indexes are able to index data sets with highly variable sampling rates, so a fixed sampling rate (that works best with conventional indexes) is no longer necessary to efficiently process data.

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 IOs 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 greatly improve on binary search or single linear interpolation. The trade-off between the query performance and memory space is discussed further in Section 5.2.

Another relevant metric for querying is the number of IOs per query. The number of IOs performed dominates the query response time and is especially important in embedded systems were predictable real-time performance is desirable. Figure 5.3 displays the average number of IOs performed per timestamp query. Binary search performs $\mathcal{O}(\log N)$ IOs and has significantly more IOs 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 IO

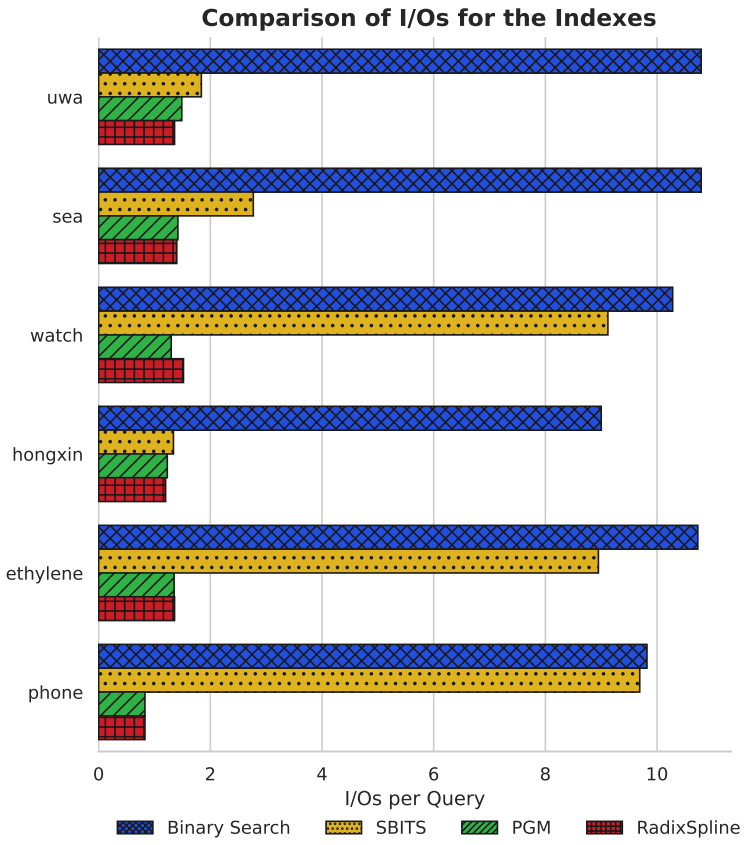

Figure 5.3: Average number of IOs per query among the indexes for each dataset in the benchmark. Lower rates are better. Learned indexes significantly reduce the number of required IOs.

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.

5.2 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 [FOL21], which prompts adaptations that trade memory and performance. The memory results are in Table 5.3.

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

Table 5.3: Memory consumption comparison among the learned indexes for each dataset in the benchmark.

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 IOs 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 IOs and lower query throughput. Table 5.4 shows statistics on the index size in bytes, IOs 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 IOs 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 IOs in a predictable fashion.

5.3 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

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

Table 5.4: Index Size in bytes, IOs per Timestamp Query, and Query Throughput (queries/sec.) for Different Error Bounds ϵ

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 IOs for writing the data pages to storage, but the index construction time may have some overhead. The insertion rates in Figure 5.4 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 throughput 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 [OKFL22a], showing that the number of IOs 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.

On embedded devices, an insertion rate of 2000 is high for the limited hardware resources available. As shown in Figure 5.4, both the learned indexes support >1951 inserts per second which represents a less than 1% overhead. This small overhead indicates that the PGM and RadixSpline learned indexes were implemented efficiently, without straining the limited processing capacity of the embedded device. 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 tradi-

tional embedded indexes such as B-Trees [OKFL22a].

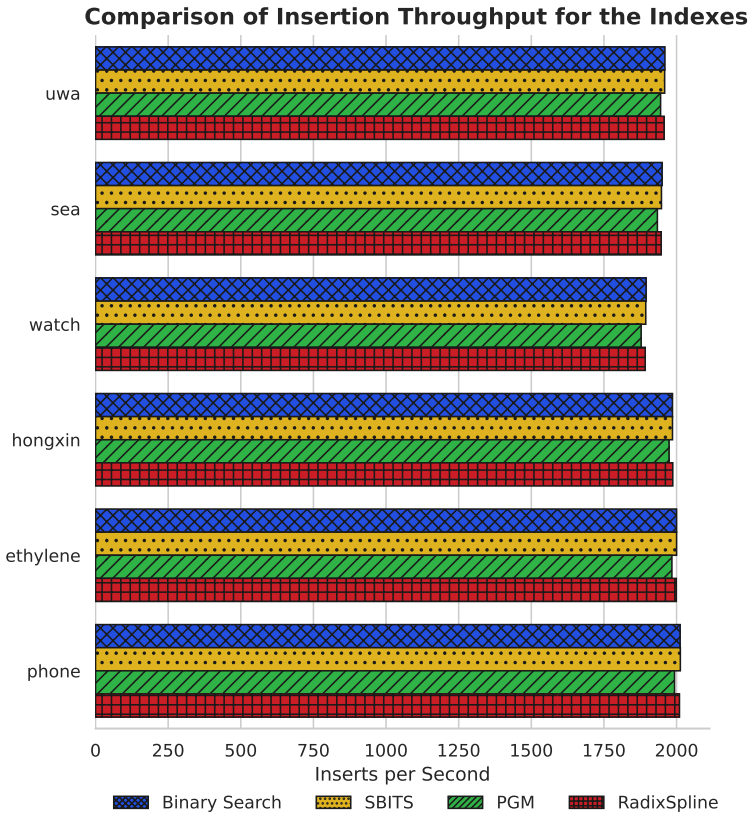

Figure 5.4: Average insertion throughput among the indexes for each dataset in the benchmark. Higher rates indicate better results. The insertion throughout is slightly lower for the PGM, but overall learned indexes remain competitive.

5.4 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 IOs per query is $2\varepsilon + 2$. 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.

Chapter 6

Conclusion

Databases employ indexes to improve query performance. Many indexes were originally developed before the prevalence of commercially viable flash storage, and thus their original implementations do not account for the hardware characteristics that flash storage exhibits.

Embedded devices use flash memory as the storage medium of choice, thus database systems running on embedded devices must also adopt design principles suitable for flash memory. These devices also have their own design considerations that must be accounted for, in addition to the considerations for flash memory.

By adapting the PGM and Spline learned indexes to support continuous appends, we demonstrate that learned indexes are well-suited for time series indexing. Learned indexes have traditionally been developed in the context of being used as in-memory indexes running on large data centers, but this work demonstrates the viability of such indexing methods on hardware-constrained embedded systems performing frequent append operations. Experimental results further express the practicality of using learned indexes in an embedded environment, where we observe lower memory footprints and IO load compared with conventional indexes.

The PGM and Spline indexes both easily fit within the 256 KB of available memory on our embedded development platform (typical of commercially available embedded devices), with 7.8 KB being the largest recorded memory consumption (used to index 100,000 entries in the *watch* dataset). As both the PGM and Spline indexes use linear approximations for their indexing structures, they also exhibit similar performance. The querying performance of the Spline and PGM indexes are within 10% of each other, with the difference mainly being due to the multi-level structure of the PGM index taking slightly longer to search. Insertion performance was also impressive, with both PGM and Spline demonstrating an almost negligible performance impact versus the simple linear index used in SBITS. This demonstrates applicability in scenarios where a high throughput is required, such as real time monitoring with high sampling rates.

6.1 Future Work

Future work will explore methods for auto-tuning the hyperparameters of the learned indexes to optimize performance with less user input. An additional challenge presents itself when considering the limited hardware resources available to embedded devices, as any tuning operation must not consume impractical amounts of memory or processing power. It may also be worth investigating whether a system can continuously monitor the pattern of the data being inserted. This could potentially allow for the system to automatically recognize when the indexing method it is currently using becomes sub-optimal and change the indexing method used for future data samples to one that is better suited to fit the detected data pattern. This functionality would also be subject to the limited hardware resources available, and care should be placed to minimize the burden on memory usage.

Bibliography

[AA14] Manos Athanassoulis and Anastasia Ailamaki. Bf-tree: Approximate tree indexing. *Proceedings of the VLDB Endowment*, 7:1881–1892, 10 2014. → pages 14

[ALAB+20] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou, Li, Andy Ly, and Christopher Olston. Learned indexes for a google-scale disk-based database, 2020. → pages 2

[BCRP16] Paolo Barsocchi, Antonino Crivello, Davide La Rosa, and Filippo Palumbo. A multisource and multivariate dataset for indoor localization methods based on WLAN and geo-magnetic field fingerprinting. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE, October 2016. → pages 64

[BFCJ+17] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. Write-optimized skip lists. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, page 69–78, New York, NY, USA, 2017. Association for Computing Machinery. → pages 16

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings - 2001 2nd International Conference on Mobile Data Management*, volume 1987, 01 2001. → pages 33

[CFT07] Abdellah Chehri, Paul Fortier, and Pierre-Martin Tardif. Security monitoring using wireless sensor networks. In *Fifth Annual Conference on Communication Networks and Services Research (CNSR '07)*, pages 13–17, 2007. → pages 27

[CJY10] Kai Cui, Peiquan Jin, and Lihua Yue. Hashtree: A new hybrid index for flash disks. In *2010 12th International Asia-Pacific Web Conference*, pages 45–51, 04 2010. → pages 17

[CMB+10] Mustafa Canim, George Mihaila, Bishwaranjan Bhattacharjee, Christian Lang, and Kenneth Ross. Buffered bloom filters on solid state storage. *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor & Storage Architectures (ADMS)*, pages 1–8, 2010. → pages 17

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979. → pages 9

[DB10] Atanu Das and Rajib Bag. Wireless sensor network based monitoring systems: A review and state-of-the-art applications. *INTERNATIONAL JOURNAL OF COMPUTER APPLICATIONS IN ENGINEERING, TECHNOLOGY AND SCIENCES (IJ-CA-ETS)*, 3:142–151, 10 2010. → pages 26

[DL14] Graeme Douglas and Ramon Lawrence. Littled: A sql database for sensor nodes and embedded applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 827–832, New York, NY, USA, 2014. Association for Computing Machinery. → pages 33, 36

[DSl+11] Biplob Debnath, Sudipta Sengupta, Jin li, David Lilja, and David Du. Bloomflash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*, pages 635–644, 06 2011. → pages 17

[EEC+09a] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proc. VLDB Endow.*, 2(1):145–156, aug 2009. → pages 43

[EEC+09b] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proc. VLDB Endow.*, 2(1):145–156, aug 2009. → pages 52

[FABM19] Athanasios Fevgas, Leonidas Akritidis, Panayiotis Bozanis, and Yannis Manolopoulos. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal*, 29:273–311, 2019. → pages 11

[FABM20] Athanasios Fevgas, Leonidas Akritidis, Panayiotis Bozanis, and Yannis Manolopoulos. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal*, 29, 01 2020. → pages ix, 19

[Fed75] Paul I. Feder. On asymptotic distribution theory in segmented regression problems– identified case. *The Annals of Statistics*, 3(1):49–83, 1975. → pages 43

[FHD+15] Scott Fazackerley, Eric Huang, Graeme Douglas, Raffi Kudlac, and Ramon Lawrence. Key-value store implementations for arduino microcontrollers. In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 158–164, 2015. → pages 34, 60

[FOKM+20] Andrew Feltham, Nadir Ould-Khessal, Spencer MacBeth, Scott Fazackerley, and Ramon Lawrence. Linear hashing implementations for flash memory. In Joaquim Filipe, Michał Śmiałek, Alexander Brodsky, and Slimane Hammoudi, editors, *Enterprise Information Systems*, pages 386–405, Cham, 2020. Springer International Publishing. → pages 9

[FOL21] Scott Fazackerley., Nadir Ould-Khessal., and Ramon Lawrence. Efficient flash indexing for time series data on memory-constrained embedded sensor devices. In *Proceedings of the 10th International Conference on Sensor Networks - SENSORNETS,*, pages 92–99. INSTICC, SciTePress, 2021. → pages 35, 40, 51, 64, 68

[FOM+19] Andrew Feltham, Nadir Ould-Khessal, Spencer MacBeth, Scott Fazackerley, and Ramon Lawrence. Linear Hashing Implementations for Flash Memory. In *21st International Conference Enterprise Information Systems Selected Papers*, volume 378 of *Lecture Notes in Business Information Processing*, pages 386–405. Springer, 2019. → pages 36

[FPL16] Scott Fazackerley, Wade Penson, and Ramon Lawrence. Write improvement strategies for serial nor dataflash memory. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6, 2016. → pages 58

[FSHM15] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 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, August 2015. → pages 64

[FV20] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020. → pages 45, 52

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, Jun 1993. → pages 8

[JRvS11] Martin V. Jørgensen, René B. Rasmussen, Simonas Šaltenis, and Carsten Schjønning. Fb-tree: A b+-tree for flash-based ssds. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, page 34–42, New York, NY, USA, 2011. Association for Computing Machinery. → pages 14

[JWZ+15] Zhiwen Jiang, Yongji Wu, Yong Zhang, Chao Li, and Chunxiao Xing. Ab-tree: A write-optimized adaptive index structure on solid state disk. *Proceedings - 11th Web Information System and Application Conference, WISA 2014*, pages 188–193, 03 2015. → pages 14

[JYW+18] Peiquan Jin, Chengcheng Yang, Xiaoliang Wang, Lihua Yue, and Dezhi Zhang. Sal-hashing: A self-adaptive linear hashing index for ssds. *IEEE Transactions on Knowledge and Data Engineering*, PP, 12 2018. → pages 17

[KBC+17] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures, 2017. → pages 2, 41, 42

[KMvR+20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2020. → pages 46, 52

[LDD11] Guanlin Lu, Biplob Debnath, and David Du. A forest-structured bloom filter with flash memory. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1 – 6, 06 2011. → pages 17

[LDM08] X. Li, Z. Da, and X. Meng. A new dynamic hash index for flash-based storage. In *Web-Age Information Management, International Conference on*, pages 93–98, Los Alamitos, CA, USA, jul 2008. IEEE Computer Society. → pages 17

[LHY+10] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3:1195 – 1206, 2010. → pages 13

[LL10] Hyun-Seob Lee and Dong-Ho Lee. An efficient index buffer management scheme for implementing a b-tree on nand flash memory. *Data & Knowledge Engineering*, 69:901–916, 09 2010. → pages 13

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, mar 2005. → pages 32

[NM08] Thomas Neumann and Sebastian Michel. Smooth interpolating histograms with error guarantees. In *Lecture Notes in Computer Science*, pages 126–138. Springer Berlin Heidelberg, 2008. → pages 52, 54, 56

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, jun 1996. → pages 13

[OFL21] Nadir Ould-Khessal, Scott Fazackerley, and Ramon Lawrence. An efficient b-tree implementation for memory-constrained em-

bedded systems. In *The 19th Int'l Conf on Embedded Systems, Cyber-physical Systems, and Applications (ESCS'21)*, 2021. → pages 37, 40

[OHLX09] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update b+-tree for flash devices. In *Proceedings - 2009 10th International Conference on Mobile Data Management*, pages 323–328, October 2009. 2009 10th International Conference on Mobile Data Management: Systems, Services and Middleware, MDM 2009 ; Conference date: 18-05-2009 Through 20-05-2009. → pages 13

[OKFL22a] Nadir Ould-Khessal, Scott Fazackerley, and Ramon Lawrence. 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, 2022. → pages 64, 70, 71

[OKFL22b] Nadir Ould-Khessal, Scott Fazackerley, and Ramon Lawrence. Performance evaluation of embedded time series indexes using bitmaps, partitioning, and trees. In Andreas Ahrens, RangaRao Venkatesha Prasad, César Benavente-Peces, and Nirwan Ansari, editors, *Sensor Networks*, pages 125–151, Cham, 2022. Springer International Publishing. → pages 51

[O'R81] Joseph O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, September 1981. → pages 52

[RKKP09] Hongchan Roh, Woo-Cheol Kim, Seung-Woo Kim, and Sanghyun Park. A b-tree index extension to enhance response time and the life cycle of flash memory. *Inf. Sci.*, 179:3136–3161, 08 2009. → pages 13

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992. → pages 39

[RPSL14] Hongchan Roh, Sanghyun Park, Mincheol Shin, and Sang-Won Lee. Mpsearch: Multi-path search for tree-based indexes to exploit internal parallelism of flash ssds. *IEEE Data Eng. Bull.*, 37:3–11, 2014. → pages 13

[RWW+17] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 125–138, New York, NY, USA, 2017. Association for Computing Machinery. → pages 21, 40

[SBB+15] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. 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, 2015. ACM. → pages 64

[TD11] Nicolas Tsiftes and Adam Dunkels. 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, 2011. Association for Computing Machinery. → pages 27, 30, 31, 32, 39

[Vig12] Stratis D. Viglas. Adapting the b+-tree for asymmetric i/o. In Tadeusz Morzy, Theo Härder, and Robert Wrembel, editors, *Advances in Databases and Information Systems*, pages 399–412, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. → pages 13

[WF17] Huiying Wang and Jianhua Feng. Flashskiplist: Indexing on flash devices. In *Proceedings of the ACM Turing 50th Celebration Conference - China*, ACM TUR-C '17, New York, NY, USA, 2017. Association for Computing Machinery. → pages 15

[WHQ+20] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. Apache iotdb: time-series database for internet of things. *Proceedings of the VLDB Endowment*, 13:2901–2904, 08 2020. → pages 23

[WKC07] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient

b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3):19–es, jul 2007. → pages 13

[Wol02] W. Wolf. What is embedded computing? *Computer*, 35(1):136–137, 2002. → pages 25

[XYLW08] Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, and Peng Wei. A reliable b-tree implementation over flash memory. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, page 1487–1491, New York, NY, USA, 2008. Association for Computing Machinery. → pages 13

[ZGD+17] Shuyi Zhang, Bin Guo, Anlan Dong, Jing He, Ziping Xu, and Song Xi Chen. Cautionary tales on air-quality improvement in Beijing. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2205):20170457, September 2017. → pages 64

[ZYLK+05] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid Najjar. 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, 2005. → pages 64