Efficient Flash Indexing for Time Series Data on Memory-constrained Embedded Sensor Devices

Scott Fazackerley, Nadir Ould-Khessal and Ramon Lawrence
University of British Columbia, Kelowna, BC, Canada

Keywords: Indexing, Database, Sensor Network, Embedded, Hashing, B-Tree, Bitmap.

Abstract: Embedded sensor devices with limited hardware resources must efficiently collect environmental and industrial time series data for analysis. Performing data analysis on the device requires data storage and indexing that minimizes memory, I/O, and energy usage. This paper presents an index structure that is optimized for the constrained use cases associated with sensor time series collection and analysis. By supporting only planned queries and analysis patterns, the storage and indexing implementation is simplified, and outperforms general techniques based on hashing and trees. The indexing technique is analyzed and compared with other indexing approaches and is adapted to all flash memory types including memory that supports overwriting.

1 INTRODUCTION

There are numerous applications for embedded systems that perform data collection and analysis including IoT devices (Adegbija et al., 2018; Al-Fuqaha et al., 2015). An ongoing trend is to perform data analysis on the data collection device as this enables faster response times and reduces network transmissions and energy usage. Data indexing for flash memory and solid state drives is a widely studied topic (Fevgas et al., 2020). However, these generic data structures based on hashing or trees are not always applicable to time series indexing on sensor devices (Qin et al., 2016) that have limited memory and query processing capabilities.

The position articulated in this work is for index structures optimized for application specific use cases for embedded devices. The focus is on small memory embedded devices that use sensors to measure and record environmental metrics, store data locally, and perform queries for aggregation and event detection. For this use case, the most important factors are accurate and efficient storage of the sensed data, and the ability to analyze and transmit this captured data. Given the limited computational power of IoT hardware (Gubbi et al., 2013), efficient use of memory is critical, and algorithms must adapt to flash storage specific constraints and performance characteristics.

The contribution of this paper is a data indexing approach optimized for sensor embedded devices, and a comparison with prior indexing algorithms. By focusing the index structure for the domain, the overall approach is simplified and results in higher performance. The technique minimizes the number of writes and the overhead in maintaining the index. A theoretical analysis and comparison is performed and demonstrates improvements over hashing and trees that are typically used.

2 BACKGROUND

Indexing on flash memory is a widely studied area, and a recent survey (Fevgas et al., 2020) overviews prior work. Index data structures have been adapted to the unique performance characteristics of flash memory such as different read and write times and the erase-before-write constraint (i.e. no overwriting or in-place writes). Common optimizations include minimizing the number of writes, performing sequential writes where possible, and exploiting hardware specific parallelism.

Significant prior work has investigated indexing on solid state drives (SSDs) as SSDs improve database server performance. Grid files (Fevgas and Bozanis, 2015), hashing (Jin et al., 2020) and B-tree variants (Kim and Lee, 2015; Kaiser et al., 2013; Hardock et al., 2019) are used. Although SSDs use flash memory, there are differences compared to indexing on embedded flash devices. Besides the flash memory chips, a SSD contains a CPU, memory, and a flash translation layer (FTL) (Chung et al., 2009). This hardware allows for I/O parallelism, and the most recent algorithms focus on exploiting this parallelism. The FTL allows algorithms to write logical pages and hides the complexities of the allocation and garbage collection of physical pages on the device.

Fazackerley, S., Ould-Khessal, N. and Lawrence, R.
Efficient Flash Indexing for Time Series Data on Memory-constrained Embedded Sensor Devices.
DOI: 10.5220/0010318800920099
In *Proceedings of the 10th International Conference on Sensor Networks (SENSORNETS 2021)*, pages 92-99
ISBN: 978-989-758-489-3

The FTL also manages wear leveling. In a server environment, the amount of RAM used for buffering is significant and allows algorithms to delay writes (via logging or write deltas) to improve performance.

In comparison, a typical small-memory embedded device may have between 4 KB and 32 KB of SRAM memory, a processor running between 16 and 128 MHz, and flash storage consisting only of raw flash memory chips. In this environment, SRAM is the most precious commodity, and many approaches designed for SSDs and large buffers are not applicable. Further, with no FTL, the algorithm must manage physical page allocation, garbage collection, and wear leveling. An additional challenge with embedded devices is that data needs to be moved to persistent storage and not left in SRAM buffers in order to not lose data as embedded devices have an increased risk of unexpected reset or power cycling.

Serially accessible flash memories are increasing in popularity for embedded devices as they require a minimum number of dedicated I/O pins. Both NAND and NOR flash are now commonly available in this format at a low price point. Each category of flash memory has its own unique attributes in terms of read, write and data access constraints. Flash memory is constructed with memory cells aligned in pages, which is the minimum write unit. Flash memory also contain an internal page buffer which holds data to be written to a page. Some devices maintain user accessible SRAM buffers to hold data but many memory devices obfuscate the internal buffer, requiring the embedded device to manage page buffers in its limited internally available SRAM. With NAND flash, data must be read from the flash memory to an internal buffer at a page level before being accessed by the embedded device whereas many NOR memory candidates support direct reads from a flash memory page.

Pages are grouped into sectors or blocks which form the minimum erasable unit in the flash memory. Some NOR flash candidates support single page erases with a higher energy and time cost. In general, data cannot be updated in place with flash memory. Regardless of the flash memory architecture, devices require erase-before-write and suffer degradation from increasing erase-write cycles which complicates data management on embedded devices (Fazackerley et al., 2016). Devices utilizing NAND flash have the additional burden of dealing with the requirement for ECC as NAND flash is prone to bit errors, while this is not required with NOR flash.

Data processing systems for embedded devices such as Antelope (Tsiftes and Dunkels, 2011) and LittleD (Douglas and Lawrence, 2014) store and process data locally, which differs from sensor network databases like TinyDB (Madden et al., 2005) and Cougar (Yao and Gehrke, 2002) where most analysis is conducted after retrieving the data over the network from the nodes. With specific focus on raw data storage and indexing, Antelope (Tsiftes and Dunkels, 2011) supported multiple index types including an inline index that stored time series data in a sorted file by timestamp. MicroHash (Zeinalipour-Yazti et al., 2005) also used a sorted time series data file, and built an index structure supporting queries by value on top of it. The advantage of storing the data as a sorted file is that variants of binary search can be used to lookup any timestamp value or range. MicroHash stored index pages intermixed with data pages. This prevents a O(1) algorithm for timestamp lookup. The value index consists of a directory of buckets, with each bucket spanning a range of values. Each index entry stores a page and offset of the record with that value. Optimizations were performed to try to completely fill index pages and minimize the number of index page writes.

PBFilter (Yin and Pucheral, 2012) minimized memory usage by sequentially writing the data and index structure. The index structure used Bloom filters to summarize page contents, and bitmap indexes for handling range queries and key duplicates. PBFilter outperforms tree and hashing techniques in terms of memory usage and write efficiency.

B-trees are common and follow one of two general approaches. Deferring writes can be done by buffering. Overwriting an existing page can be avoided by logging changes either in another area on flash or in special areas of the page itself (Kim and Lee, 2015). Logging and write defer strategies rely on sufficient memory to buffer modified pages so that multiple updates can be batched together in one physical write. Write-optimized trees (Bender et al., 2015) buffer modifications and write them out in batches to amortize the write cost and avoid small random writes.

Overwriting an existing page in flash memory is allowed (Kaiser et al., 2013; Fazackerley et al., 2016; Hardock et al., 2019) in certain cases. This allows for different techniques for erase and write avoidance and modifications of data structures to improve the performance of some techniques. There are challenges related to error correction and the type of writes that support overwriting for NAND flash memory.

3 PROBLEM DEFINITION

An embedded device performs a fixed set of data operations, unlike a general database server. When the device is designed and deployed, these data process-

ing operations will be known and implemented in the code base. By exploiting this knowledge, it is possible to build more implementation-specific optimized index structures.

The data set is an append only time series where each entry consists of a timestamp and one or more sensor values. A sensor value is produced by reading from a sensor connected to the embedded device, and is an integer of size between 8 and 32 bits. When reading using an analog-to-digital converter, the raw sensor value is then converted into an actual value based on a sensor specific transfer function. In many cases the value produced is a floating point value. Processing and storing floating point values on many small embedded devices increases device overhead due to architectural limitations. Using the value in its discrete raw form instead of the converted value is more efficient.

The embedded device has these properties:

- Memory-constrained (4 KB to 128 KB)
- Flash memory storage (may be NOR or NAND)
- May have either raw or FTL-based memory
- Flash memory may support page-level overwriting and/or byte-level direct reads.

The embedded environment characteristics are known and available to the algorithm including:

- The number of sensors and the frequency of sensor measurements
- The size of the flash memory available to be used and its characteristics
- The data processing requirements including queries, sliding window aggregation, and consistency requirements
- Append only data with increasing timestamps

For consistency requirements, the system must support both page level and record level consistency. The types of queries supported include:

- Equality and range timestamp queries
- Equality and range value queries (but may be restricted to certain value ranges of interest)

These requirements are reasonable given that sensor devices are developed, configured, and deployed with specific use cases and require efficient and reliable operation for extended periods of time.

4 INDEXING APPROACH

The proposed indexing approach minimizes writes and memory usage. The index structure adapts to the known query requirements and optimizes for the queries specifically required for a particular use case.

4.1 Data Record Storage

The foundation of the approach is that time series data records are written sequentially in pages to flash. This is similar to MicroHash (Zeinalipour-Yazti et al., 2005) except that all data pages are contiguous, and are not intermixed with index pages.

Storing data pages contiguously works regardless if the flash has an FTL or supports overwrites. If the flash memory has an FTL, then all writes are logical writes and all physical allocations are hidden. If the flash does not have an FTL, there are several implementation possibilities:

- Page level consistency - buffer page in memory and only write out full pages to flash
- Record level consistency:
  - Overwrites supported - append record to end of page using overwrites
  - No overwrites supported - write page to free space at end of memory region temporarily, when page is full, write to end of data area

The ability to store the data records in sequential order by timestamp enables efficient timestamp searches and minimizes the number of writes.

4.2 Design Optimized Value Indexing

Supporting value based queries efficiently requires adding an index structure on top of the raw data storage. Without an index, a query on a data value requires a sequential scan of all data pages.

The design decision is to deploy indexes based on the types of queries that will be required by the embedded device application. Since the device will not be executing interactive queries, these queries are known during design time. The queries generally fall in two categories:

- Point and range outlier value detection - value $\geq V$ or value $\leq V$ for some value V
- Recent history sliding window aggregates - for the most recent T time units, return the average, max, min, or count of value(s)

A key observation is that the integer values often have a limited domain (e.g. 10 bits) and demonstrate skew and temporal locality. For example, a sensor collecting temperature data will have the values change relatively slowly with a regular trend.

To support efficient aggregate queries, each data page header contains the smallest and largest timestamp values, the count of the number of records, the minimum and maximum values in the page, and the sum of all values in the page. The page header also stores a small bitmap index that approximates the data distribution. The size of the bit vector is determined based on design requirements. Each bit represents the presence of a value in the particular range with ranges being user defined.

An example page structure that uses 2 byte values is in Figure 1. The page structure used for any particular time series data set depends on the queries that must be supported. The only required fields are the 4 byte page id and 2 byte count. The time series range, aggregate summary and bitmap are optional. The overhead is about 24 bytes or 4.5% of a 512 byte page. To support multiple different values, the aggregate summary and bitmap is repeated for each value in the sensor record.

Figure 1: Page Structure.

The bitmap is designed based on query requirements. For example, consider a value range of 0 to 100. One possible bit vector may divide the range into 10 equal buckets (0-9, 10-19, ..., 90-100) and require 10 bits of storage. If the values of interest are not equally distributed, the bit vector can be defined to capture the data of most importance. For example, if the analysis queries are looking for very small or very large data values, it may be better to use only 4 bits with the ranges (0-9, 10-49, 50-89, 90-100). At the limit, a single bit indicator can be used to determine a range of interest. For example, if the queries are focused on only values $\geq V$, then 1 bit is used and is set only if the page contains values $\geq V$.

Database index tuning is commonly performed in database systems, but varying an embedded index structure based on the required data analysis is unique. The bit vector filter in the header is configured based on design specifications and only requires some additional space in each page. No additional I/Os are required. However, by itself it does not significantly improve query performance or reduce I/Os, as unless direct byte reads are supported, the system still must read the entire page from flash (although the bit vector does save time in processing records in the page once it is read).

The bit vector for each page is also written out to an index area in a separate area of the flash memory. Thus, index pages will not be intermixed with data pages. One buffer is used to store the latest index page, and it is only written out when full. Given that a typical bit vector size per page is between 4 and 16 bits, for a 512 byte page, the index size overhead and additional I/Os is between 0.1 and 0.4%.

4.3 Querying by Timestamp

The pages are ordered by timestamp, and any timestamp can be retrieved in $O(1)$ reads by calculating an offset using the start timestamp, the query timestamp, and the rate at which records are written to flash (based on sampling rate and number of records/block). A given query timestamp t_q can be located by calculating its page index:

$$\text{page index} = (t_q - t_s) * \frac{r}{B} \quad (1)$$

The first timestamp value is t_s, and r is the sampling rate in records/second with B as the number of records/page. The page index is converted to a physical offset by multiplying by the page size in bytes.

Since the memory is treated as a circular storage, after the data fills the memory for the first time, it is necessary to wrap the offset value based on the current physical offset of the first data page in memory.

For example, assume that the memory can store up to 100 data pages, and the data has wrapped around such that the current first data page is at offset 50. If the page index is calculated as 75, then the physical page offset is $(50+75) \mod 100 = 25$.

4.4 Querying by Value

For queries by value of the form $v_1 \leq value \leq v_2$, query performance depends on the selectivity of the query predicate, s, and the ability to capture that selectivity in the bitmap index. Every index based query must read the entire index, which costs approximately $0.001 * N$ where N is the number of data pages. This is a sequential read and requires only one buffer for the current index page.

If the query predicate selectivity is s, then on average $s * N$ data pages will be read to process the query. The actual number of page reads depends on how the records matching the query predicate are distributed across the pages. If the query predicate exactly aligns with the bitmap vector such that no false positives are generated, then the algorithm will read the minimum number of data pages required and is optimal. Since the indexes can be constructed to support specific queries, it is possible to get perfect matching and no false positives subject to space limitations. The algorithm for index querying by value is in Figure 2.

When comparing the index and query bitmap, the two bit vectors can be combined with logical AND,

```
Convert query predicate into query bitmap
Let P = 0 be the current data page index

for each index page
  for each index record
    if index bitmap AND query bitmap > 0
      read data page for this index record
      for each record in data page
        output record if predicate is true
```

Figure 2: Querying by Value Range.

and any value greater than 0 (i.e. any bit set) indicates that the data page should be read.

The overall cost of the algorithm is the cost to read the index pages and any data pages that contain records that match the predicate. If the bitmap index supports the query predicate precisely, the minimum number of data pages, $s * N$, are read.

As an example, consider a query for $value >= 90$. In the previous section, two possible bitmaps were defined one with 10 bits and the other with 4 bits. In either case, the last bit in the vector, matches the predicate exactly, and only pages with records that match the predicate will be read. On the other hand, if the query is $25 \leq value \leq 75$, then both bitmaps will return false positives. The 10 bit vector will have fewer false positives as its value ranges have finer granularity and better overlap with the query predicate.

The other common query is sliding window summaries. The page-level aggregate summaries support aggregate calculations without processing the data records, and the filtering of rows by examining the minimum and maximum data range. Using the header summary is especially useful if the memory supports byte level or partial page reads so only the header information is read rather than the entire page.

In most use cases, the query will not scan the entire data set but only the last few pages which represent the latest data in the sliding window. Given sufficient memory, previous data pages are obvious choices for buffering in RAM. If the number of pages in the sliding window is small, then it is more efficient to read the pages rather than building another index of the summary values.

4.5 Memory Management

The minimum memory requirement is one write buffer for the current data page and a read buffer to read the data records. The addition of the bitmap index requires an additional write buffer for the index, and a read buffer when processing index queries. Any additional memory available would be used to buffer the most recent data pages, if the system uses sliding window queries on recent data. Thus, the minimum memory requirement is 2 buffers without bitmap indexing and 4 buffers with bitmap indexing.

4.6 Handling Variability

Design and operational parameters may change throughout the life of an embedded sensor device. These changes include adding or removing sensors, modifying the frequency of sensor readings, and adapting the queries performed on the device.

The approach easily handles changes in the sensors and their sampling frequency. These changes only impact querying by timestamp, as timestamp record locations are calculated based on sampling frequency and records per page. These parameters are known at any given time, and if they change during run-time, the system must only store a history of previous values and the time when the change occurred.

For example, consider that the system has already stored 50 data pages with the frequency r as 10 samples/hour and $B = 10$. If r is changed to 20 samples/hour, then the system must record that starting at page offset 50, r is 20. To calculate the page index for a time 120 hours from the first timestamp:

$$\text{page index} = 50 * \frac{10}{10} + (120 - 50) * \frac{20}{10} = 190 \quad (2)$$

Thus, given knowledge of past sampling rates, calculating the page index remains straightforward. Supporting different bit vector indexing strategies is also possible by recording the page indexes when the bit vector structure was changed.

4.7 Wear Leveling

Wear leveling is performed by treating the memory as a circular array and wrapping around to the front of the memory region using a modified Frontier Advance Wear Leveling approach *(FAWL)*. Figure 3 shows the layout of memory for wear leveling. Pages are written to the location tracked by the *page write frontier*. In advance of the page write frontier, the system maintains a region of erased pages *(write frontier extent)* that allows pages to be written to the device. The *temporary write extent* is located in front of the write extent and is used as temporary record level storage. The size of the temporary write extent is a multiple of the memory device's minimum erase unit and is aligned on a sector erase boundary. General sector erases are performed at the *sweep frontier* in front of the active data extents.

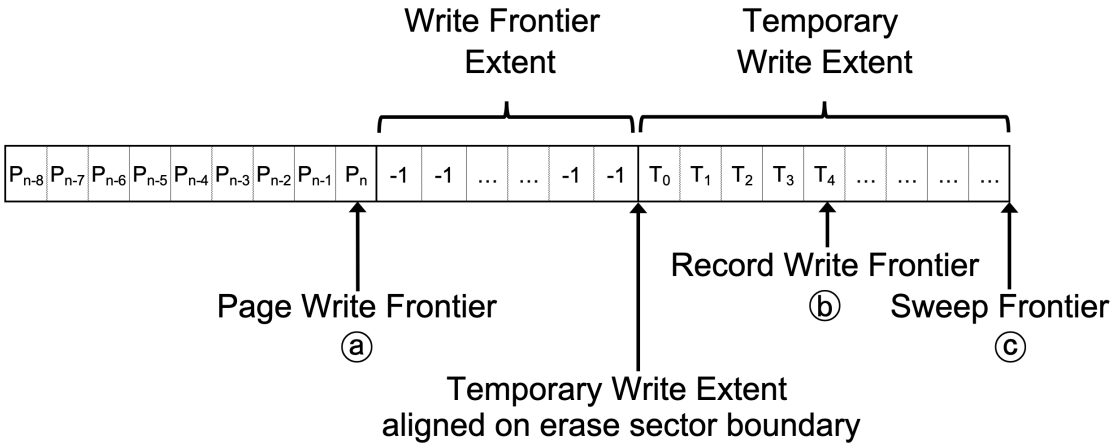

Figure 3: Frontier Advance Wear Leveling.

If page level consistency is used, complete pages will be written to the page at the page write frontier (Ⓐ). The system assigns an increasing page id to each page written (P_n) which is used to track the age of pages in the system. If record level consistency is being used, record updates will be written to pages (T_n) inside the temporary write extent at the record write frontier (Ⓑ). When enough records have been written to the temporary write extent to fill a page, the full page is then written to the page write frontier.

As pages are written to the system and the page write frontier advances towards the temporary write extent boundary, the temporary write extent will be advanced to the next sector of free pages. The system will erase the sector tracked by the sweep frontier (Ⓒ), advance the temporary write extent and record write frontier to the position of the sweep frontier, and the sweep frontier will then be advanced to the next sector that will be targeted for erase. The stale temporary write extent will then be erased. The write frontier extent will then be updated to include this reclaimed area.

As *FAWL* guarantees that the timestamp records are written to the memory in a linearly increasing fashion, the sweep frontier will always reclaim the oldest records in the system. Additionally, it wears memory uniformly by writing and erasing sequentially, eliminating the need for a complex block write/erase management scheme or FTL.

The system records the current page index for the start and end of the valid data pages. Preferably, this information is stored in non-volatile memory in case of system failure. Otherwise it is recoverable by scanning the memory. The first data page has the smallest page id, and the last data page has the largest.

5 ANALYSIS

The sequential bitmap indexing for time series technique, referred to as SBITS, is compared to other indexing approaches for embedded systems. A summary of the approaches is in Table 1.

The inline index developed for Antelope stored data records in increasing order by timestamp. Searching by timestamp used binary search, but searching by value is not supported. The minimum memory is 2 pages (one write buffer for data records and one read buffer for queries). This index structure has low overhead, but does not also support efficient value queries.

The closest comparable strategy is MicroHash that has a sequential data file intermixed with index pages. Each index page stored index records for a particular data value. The minimum memory requirement for MicroHash was 6 pages (write buffer for data, write buffer for index, 2 read buffers (index and data), buffer for directory index, and buffer for root block). MicroHash also uses buffers for caching index pages.

The MicroHash index size varies between 10% and 50% of the data size and depends on the size of the data records, the variability in the data, and the amount of memory available to cache index blocks. By intermixing data pages and index pages, a modified binary search was required in MicroHash, which on average required about 5 reads for the data sets tested. Further, the index consumed more space as the index records are larger and could not always completely fill the pages. This is due to MicroHash using in-memory hash partitioning of the data values in the index and being forced to flush an index page to storage whenever space must be freed up.

The key difference and limitation of MicroHash

Table 1: Comparative Analysis.

Algorithm	Index Size	Min. Memory	Timestamp Query I/Os	Value Query I/Os
SBITS	$0.004 * N$	2 to 4	1	$0.004 * N + s * N$
Antelope index	0	2	$O(log_2 N)$	N/A
MicroHash	$0.1 * N$ to $0.5 * N$	6	5	$0.1 * N + s * N$
PBFilter	$\frac{B*N}{I} + \frac{N}{F}$	4	$O(log_2 N)$	$\frac{N}{F} + s * N$
B-tree	$\frac{B*N}{k}$	5	N/A	$O(log_k N) + s * N$
Linear hash	$\frac{B*N}{I}$	3	N/A	$O(1)$ (point query)

compared to SBITS is that index records based on offsets consume more space than the bitmap index, and more importantly, good performance is only achievable if you can buffer one page for each bucket in the hash table. If the hash table has 4 buckets, then that would require 4 in-memory buffers. In some data sets with low variability, it is possible to have fewer memory buffers, but this causes issues with only partially filled pages. The write cost using this index technique is considerably higher. When reading, more buffers are required for range queries as each bucket that intersects the query range needs a buffer in order to make sure the data file is not read multiple times.

PBFilter writes data and index pages sequentially and uses bloom filters and bitmaps for data summaries. PBFilter requires at least 4 memory buffers and has more writes as it has both a record-level index and index summaries. The index size consists of the space used to store an index entry for each data record (where I is number of index entries that fit in a page) plus space used to store the summary of each data page (where F is number of bloom filters that fit in a page). Given that an index record size may be near the same size as a data record (i.e. 12 bytes), the index size may be as large as the data size. PBFilter does not discuss timestamp based queries on the data file, but given that the data is written sequentially binary search is possible. Value queries require reading the summary index and then any matching data pages. Compared to SBITS, PBFilter uses more memory and requires more space and writes to maintain the index.

B-tree implementations on embedded devices are restricted on RAM so techniques based on extensive buffering are not applicable. A B+-tree storage structure will allow for efficient queries on either the timestamp or value but not both. Assuming the data is stored sequentially by timestamp, then the B+-tree is built on value. If the number of index records per page is k, search and insert is $O(log_k N)$. Buffers required are 2 for data (read and write) and 2 for B+-tree (minimum of 2 for supporting split). To use any form of log buffering, additional buffers are required. At minimum, a write buffer is required for log records. LSB-tree (Kim and Lee, 2015) is executable on memory-constrained devices as it uses a single log area to support updates and requires only one additional buffer for the log file. However, it generates more writes and has costly re-organization operations to rebuild the tree periodically. Query performance and index size depends on k. B+-tree implementations are not optimized for append only time series data.

Linear hash implemented on embedded devices (Feltham et al., 2019) supports key based lookup in constant time. The minimum memory usage is 3 buffers, but better performance requires a buffer per hash partition otherwise every insert requires a page to be read from memory and then written with the new data record. Timestamp based queries and range queries on value are not supported.

Overall, the fundamental position taken is that the index structure can be adapted at design time given that the queries to be supported will be known in advance. With append only time series data and known query patterns, SBITS minimizes I/Os and energy usage by building the minimal possible index to support the required queries. The number of writes is minimized, and the index size is less than 1% of the data size. Queries on timestamp require only 1 read, and value queries will read the minimum amount of data if the bitmap index is tuned to match query predicates.

6 FUTURE WORK AND CONCLUSIONS

Efficient data storage and indexing in embedded systems is critical as these systems collect and process more data. This work presented the SBITS approach that is optimized for append only, time series data sets. An important insight is that embedded systems typically do not process general queries, and knowledge of data analysis requirements allows for the index structure to be more efficient. The approach was compared with other indexing approaches using hashing and trees and shown to minimize memory usage while allowing for highly efficient timestamp and value queries. Future work will experimentally evaluate the approach on various hardware systems.

REFERENCES

Adegbija, T., Rogacs, A., Patel, C., and Gordon-Ross, A. (2018). Microprocessor Optimizations for the Internet of Things: A Survey. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(1):7–20.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys - Tutorials*, 17(4):2347–2376.

Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. (2015). An Introduction to Bε-trees and Write-Optimization. *Usenix Mag.*, 40(5).

Chung, T.-S., Park, D.-J., Park, S., Lee, D.-H., Lee, S.-W., and Song, H.-J. (2009). A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332 – 343.

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

Fazackerley, S., Penson, W., and Lawrence, R. (2016). Write improvement strategies for serial NOR dataflash memory. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6.

Feltham, A., Ould-Khessal, N., MacBeth, S., Fazackerley, S., and Lawrence, R. (2019). 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.

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

Fevgas, A. and Bozanis, P. (2015). Grid-File: Towards to a Flash Efficient Multi-dimensional Index. In *Database and Expert Systems Applications*, pages 285–294. Springer.

Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.*, 29(7):1645–1660.

Hardock, S., Koch, A., Vinçon, T., and Petrov, I. (2019). IPA-IDX: In-Place Appends for B-Tree Indices. In *15th International Workshop on Data Management*, pages 18:1–18:3. ACM.

Jin, P., Yang, C., Wang, X., Yue, L., and Zhang, D. (2020). SAL-Hashing: A Self-Adaptive Linear Hashing Index for SSDs. *IEEE Trans. Knowl. Data Eng.*, 32(3):519–532.

Kaiser, J., Margaglia, F., and Brinkmann, A. (2013). Extending SSD lifetime in database applications with page overwrites. In *6th Annual International Systems and Storage Conference*, pages 11:1–11:12. ACM.

Kim, B. and Lee, D. (2015). LSB-Tree: a log-structured B-Tree index structure for NAND flash SSDs. *Des. Autom. Embed. Syst.*, 19(1-2):77–100.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173.

Qin, Y., Sheng, Q. Z., Falkner, N. J., Dustdar, S., Wang, H., and Vasilakos, A. V. (2016). When things matter: A survey on data-centric internet of things. *Journal of Network and Computer Applications*, 64:137 – 153.

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

Yao, Y. and Gehrke, J. (2002). The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Rec.*, 31(3):9–18.

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

Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. (Dec 13, 2005). MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*, pages 31–43. USENIX Association.