# EmbedDB: A High-Performance Time Series Database for Embedded Systems

Justin Schoenit[a], Seth Akins[b] and Ramon Lawrence[c]
*University of British Columbia, Kelowna, BC, Canada*

Keywords: Database, Key-Value Store, Embedded, Arduino, Internet of Things, SQL, Time Series, SQLite.

Abstract: Efficient data processing on embedded devices may reduce network communication and improve battery usage allowing for longer sensor lifetime. Data processing is challenged by limited CPU and memory hardware. EmbedDB is a key-value data store supporting time series and relational data on memory-constrained devices. EmbedDB is competitive with SQLite on more powerful embedded hardware such as the Raspberry Pi and executes on hardware such as Arduinos that SQLite and other previous systems cannot. Experimental results evaluating EmbedDB on time series query processing show a speedup of five times compared to SQLite on a Raspberry Pi on many queries, and the ability to execute data processing on small embedded systems not well supported by existing databases.

## 1 INTRODUCTION

Embedded databases are used in a wide variety of environments from supporting desktop and mobile applications to use in custom embedded and sensor systems, including the Internet of Things (IoT). Due to this diversity of applications and hardware capabilities, there are numerous systems used with different architectural choices and supported features. Higher performing devices such as cell phones and the Raspberry Pi have the hardware and operating system capabilities to run embedded SQL databases such as SQLite (Gaffney et al., 2022). Embedded and sensor devices have significantly less hardware resources and operating system support, which restricts the database systems they are able to run.

For embedded devices, the key goal is high performance despite significant hardware limitations that result in many systems being unable to execute on the smallest devices as either they require too much RAM, code space, or operating system support. Prior research has produced systems minimizing memory usage in this environment (Anciaux et al., 2003; Bonnet et al., 2001; Douglas and Lawrence, 2014).

Our contribution is EmbedDB, an embedded key-value, time series database with high performance on embedded devices, that is flash-optimized, has no code dependencies, does not require an operating system, and functions with only 4 KB of RAM. EmbedDB is primarily optimized for storing and querying time series data collected by sensor nodes. Experimental results demonstrate that EmbedDB outperforms SQLite on a Raspberry Pi for many queries and on smaller embedded devices its read and write performance maximizes hardware capabilities.

[a] https://orcid.org/0009-0000-1791-3437
[b] https://orcid.org/0009-0001-8346-3590
[c] https://orcid.org/0000-0002-6779-4461

## 2 BACKGROUND

### 2.1 Embedded Systems

There is a wide range of devices performing data processing. Cellphones, tablets, and other mobile devices have significant CPU, memory, and storage capabilities and run an operating system such as iOS or Android. These devices can run relational databases with the most commonly deployed system being SQLite (Gaffney et al., 2022). The Raspberry Pi (Raspberry Pi Foundation, 2023) runs a Linux variant with SD card flash storage and can execute databases such as MySQL, PostgreSQL, or SQLite.

Embedded processors may be deployed in many industrial, environmental, and system sensing and monitoring applications. These devices trade-off performance to minimize cost and may have a CPU clock



Schoenit, J., Akins, S. and Lawrence, R.
EmbedDB: A High-Performance Time Series Database for Embedded Systems.
DOI: 10.5220/0012558100003690
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024) - Volume 1*, pages 240-249
ISBN: 978-989-758-692-7; ISSN: 2184-4992
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

speed from 8 to 256 MHz and between 4 KB to 512 KB SRAM. Given the hardware resources, small devices do not typically run Linux and only the highest performing run some form of embedded OS (Javed et al., 2018). SQLite and many other embedded databases do not run on smaller hardware as they require more SRAM and support for the Linux I/O API.

Embedded devices have features that affect data processing. Data storage is typically on flash memory either raw NOR or NAND chips or an SD card. Flash memory writes are more time-consuming than reads, and algorithms perform more reads to avoid writes. Most flash memory is not byte addressable, so bytes are organized into pages. Flash memory requires that a page must be erased before writing, which prevents modifying data in place. Instead, the page is read from the device, modified and then written back to an erased location. Device wear occurs during every erase/write operation, which may result in failure and data corruption over time.

SD cards have a page-level abstraction layer. References to logical pages are converted to physical addresses by a flash translation layer (FTL). This translation simplifies implementation by ignoring physical storage characteristics. The FTL handles logical page overwrites by selecting a new physical page in storage and updating its page translation table. The FTL ensures pages are written evenly throughout storage. Without an FTL, the database must handle writes, erases, and memory management internally.

### 2.2 Embedded Data Processing

Time series data sets are commonly collected by embedded devices and consist of records where each record has a timestamp and one or more data values collected at that timestamp.

Data processing on embedded systems has been performed with two general approaches. Research on developing sensor networks such as TinyDB (Madden et al., 2005) had each embedded, sensor device operating as a data source that was managed and queried by a controller that performed the query optimization and planning. In this architecture, the focus is on efficient query planning and data routing.

Other prior research developed indexes and data systems for execution on embedded systems with focus on improving the time and energy efficiency for local data processing. The goal is to provide a lightweight database abstraction layer that is an alternative to developers building their own data storage code.

Embedded data systems include Antelope (Tsiftes and Dunkels, 2011) and LittleD (Douglas and Lawrence, 2014). These systems support some form of relational query processing and a limited form of SQL. Challenges include parsing and optimizing SQL queries given low memory and efficiently executing relational operators.

Antelope (Tsiftes and Dunkels, 2011) stores time series data in sorted order and uses a binary search for record retrieval by timestamp. MicroHash (Zeinalipour-Yazti et al., 2005) uses binary search for timestamp queries and range partitioning to index data values. Bloom filters are used to summarize and index data in PBFilter (Yin and Pucheral, 2012). SBITS (Fazackerley et al., 2021) is a sequential indexing structure supporting timestamp queries using linear interpolation and data queries by using a user-customizable bitmap index. Experimentation (Ould-Khessal et al., 2022) showed that SBITS outperforms MicroHash and other embedded indexing techniques including B-trees (Ould-Khessal et al., 2021).

Improved time series indexing is achieved by using multiple linear approximations to predict a record location based on its timestamp. Applying learned indexing to embedded time series indexing demonstrated (Ding et al., 2023) significantly faster record lookup while only consuming about 1 KB of RAM. The RAM is used to store the linear approximation information including the starting point and ending point of each line. The learned index approach is effective when the data set is not as regular and cannot be easily approximated by a single linear function. Events such as changing the sampling frequency, power failures, or sensor outages cause the data set to be less regular and index performance to degrade.

Overall, there is still a need for a data system as simple as SQLite for small embedded systems.

## 3 EmbedDB ARCHITECTURE

EmbedDB is a database system that supports both time series and relational data as key-value records. The architecture for EmbedDB is in Figure 1 and its features are:

- Supports small devices with a minimum memory requirement of 4 KB
- Optimized insert performance for time series data
- Customizable key (Ding et al., 2023) and data indexes (Fazackerley et al., 2021)
- Generic storage support including SD cards and NOR/NAND chips
- No library or operating system dependencies
- Advanced query API and SQL to C translator for SQL queries

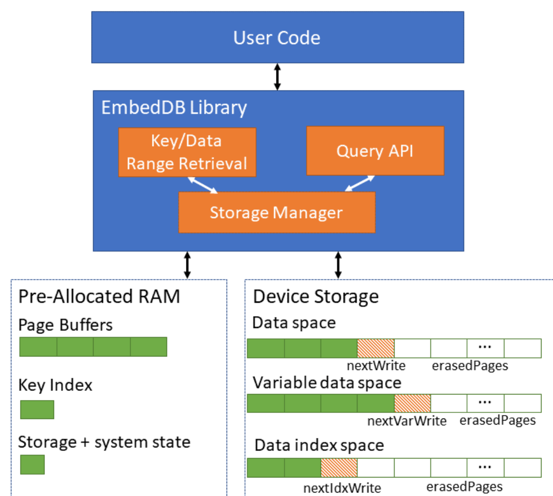


Figure 1: EmbedDB Architecture.

## 3.1 Storage

A developer using EmbedDB pre-allocates all memory used by the system including page buffers, the key index, and the storage and system state. For the most basic insert functionality, a minimum of 2 page buffers are used that buffer the current page in memory before writing to storage and a page is used for reads. With physical device pages often being 512 bytes, the minimum page buffer size is 1 KB. The C struct maintaining the storage and system state occupies 200 bytes. The optional key index has variable size depending on user requirements but is often below 1 KB. In Figure 1, four page buffers are shown, which is the common configuration. The other two page buffers are used for reading and writing the data index, which is an optional component to speed up data value searches.

The storage manager supports any storage device that is page accessible. For SD cards with a file interface, the storage manager interacts using the SDFat library[1] and does not manage the physical memory. For raw NAND, NOR, and DataFlash memory without an FTL, the developer provides EmbedDB with a memory address range, and EmbedDB manages the memory space as a circular queue. There is a current write location where the next write will be performed. After that location, the system guarantees that pages are erased by performing block-level erases. Treating the storage as a circular queue enables developers to set bounds on the amount of storage consumed, and the system will guarantee that the latest data is always preserved. It also ensures consistent device wear and inserts are performed sequentially for higher performance. If the storage space is full, the system overwrites the oldest data automatically or can be configured to generate an error and stop adding new data.

The base data storage format consists of fixed-size key-value records allocated as a sequential time series index (Fazackerley et al., 2021). The data records are key and value byte arrays. The key array may be 4 or 8 bytes. The size of the data value array is limited by the device page size. Using fixed-size records enables efficient storage supporting the most common use case of small time series records storing a timestamp and one or more numeric data values.

To support larger records and variable-sized fields, an optional variable data storage is supported, shown as *variable data* in Figure 1. Each data record may link out to the variable-storage memory area. This enables support for larger text fields and images. The variable data space is also managed as a circular memory range that is distinct from the data space.

## 3.2 Inserts

EmbedDB is optimized for low-overhead, high-performance inserts. When a record is inserted, it is stored in the current write page buffer in RAM. When a page is full, it is written to storage in the data space at the *nextWrite* location. If the record has variable data, a separate page buffer is used for buffering the variable data before writing to the variable data space.

Data pages are not overwritten in place. All page writes are performed on the next erased page in sequential order. With no overwriting, there is no need for write-ahead logging to handle failure. On failure, the system scans the data space to find the last active page and *nextWrite* location and proceeds as usual.

## 3.3 Indexing

The system supports two types of indexes that can be optionally added to the base record storage. If no indexes are included, key lookup queries are performed using binary search, and queries examining the data values require a scan of the entire data file.

### 3.3.1 Key Index

Key indexing is performed using a spline learned index (Ding et al., 2023) that uses a small amount of memory to store a sequence of linear approximations that map a record key (timestamp) to a page location in storage. An example is in Figure 2.

Whenever a data page is written to storage, a key index record is added consisting of the minimum key on the page and the page number. In Figure 2, each

[1]https://github.com/greiman/SdFat

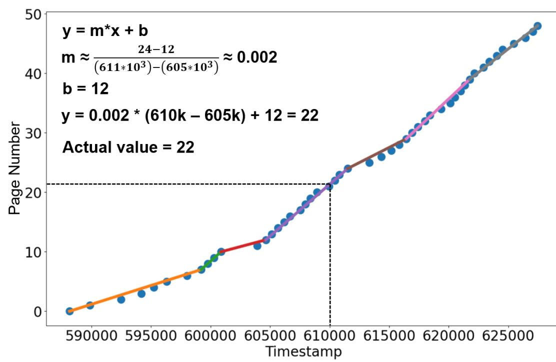


Figure 2: Key Index using Linear Approximations.

blue dot represents a data page number and the minimum key (timestamp) record on that page.

The spline method for building linear approximations starts the first line with the first data point. Whenever a data point is added, it checks to see if that point is within the given error, ε, specified by the user from the current linear approximation. If it is within the error bound, the point is discarded and no changes are made. If it is not within the error bound, the current linear approximation is ended at the previous point, and this previous point is the start point for the next linear approximation.

In Figure 2, each color line represents a distinct approximating line segment (total of 7). For each line segment, the data stored is the start point of the segment consisting of key and page number. The end point is not stored as it is the same as the starting point of the next line segment. For this example, the key index occupies (7+1) points * 8 bytes/point = 64 bytes. In general, the memory consumed depends on the number of keys, the variability of the keys, and the user specified error rate tolerance, ε.

A developer provides EmbedDB with a pre-allocated memory area to use, and EmbedDB manages the index to fit in that memory area. When the memory is full, some spline points are dropped which lowers query performance for older data ranges.

For queries on a given key, the table storing the spline points is searched for the segment start point just below the key value. Then its linear formula is used to approximate the page location of the key and provide a search range of $(loc - \varepsilon, loc + \varepsilon)$. Figure 2 shows searching for key 610,000.

#### 3.3.2 Data Index

The user may optionally add a data index to speed up queries on a data value. The data index is implemented using a bitmap filter that is customizable by the developer. Similar to (Fazackerley et al., 2021), the data index consists of a bitmap computed for each page by a user-supplied function. The size of the bitmap is configurable and typically between 1 and 4 bytes. When a page is written, the bitmap function is applied for every record in the page and the bitmap updated. The bitmap is stored in the data page as well as on a data index page, which is written to storage when full.

For querying, EmbedDB reads the pages in the data index sequentially and processes each bitmap contained within. A user query specifying a particular data range is converted into a query bitmap. If a page bitmap overlaps with the query bitmap, the page is read, otherwise it is skipped. Although EmbedDB must perform a scan of the data set to process a data value query, it is able to avoid many I/Os by using the bitmap to only retrieve pages from storage that potentially have matching data values. The data index reduces the amount of data scanned based on the selectivity of the query and how accurate the bitmap functions capture the data distribution.

For example, consider a case where the data values have only 8 possible values. An 8-bit bitmap can be used to indicate the presence of each value in a page. The number of pages read will depend on how the data values are present in the pages. In the best case, only one data value is present in any given page. In that case, only 1/8 of the data pages are read when searching for any particular data value as the bitmap will indicate the pages that contain the queried data value. In the worst case, all data values appear on every page, and every page will be read.

In practice, the number of data values are much larger than the number of bitmap bits, so each bit will represent a range of data values. The bitmap index is highly effective for sensor data as data values tend to be clustered based on the sensor measurements taken. An advantage is that no random writes are performed during record insert, and no random reads are performed during queries.

### 3.4 Query Processing

Figure 3 contains a code example on how EmbedDB is used to insert records and then perform a query on a given key and/or data range. Records are inserted as key-value byte arrays, and are retrieved using the iterator. The example shows a query with key range 100 to 1000 and for data values >= 50. If the user supplies a key value or range, then the key index is used to retrieve the first record and then iterate records sequentially. If a data range is also specified, the data filter is applied by the iterator before returning records to the user. For queries involving a key, the data index is not used. For queries involving only a data range,

the data index is used (if available) to only read pages that potentially may have values in the data range as indicated by the bitmap index.

Single key lookups are often performed with just one I/O using the key index. The performance of key and data range queries depends on the selectivity of the query and the number of pages scanned.

### 3.5 Advanced Query API

Often data analysis requires more than key or range lookup. EmbedDB implements an advanced query API allowing support for queries typically done using SQL. The advanced query API builds upon the query iterators to support common relational operators such as grouping, aggregation, filtering, and projection. A developer supplies a function to the core operators to perform. Operators may be connected together in iterator form in complex execution trees. Users can use the `exec()` function to execute arbitrary code over top of an iterator. Example queries supported by the API are in Table 3. The code and functions are not shown due to space limitations.

### 3.6 SQL Pre-Compilation

Many users like the convenience of SQL for specifying queries and the optimization performed by SQL databases. Prior work has tried to implement a limited SQL interface for embedded systems (Douglas and Lawrence, 2014; Tsiftes and Dunkels, 2011), but there were issues in the amount of SQL that can be supported and the associated overhead. The challenge is that parsing a string SQL statement and performing optimization requires a significant amount of memory that the devices do not have. Further, most embedded systems have very well-defined data processing requirements such that it would be extremely rare to have dynamic, user-specified queries that are unknown to the system before development. This is significantly different than databases on servers.

Given these domain constraints, EmbedDB supports SQL using a pre-compilation tool. The developer specifies the structure of their key-value store using a SQL CREATE TABLE statement as well as CREATE INDEX statements if there is an index on any data column. The developer then provides the SQL query that they want to support. The SQL pre-compiler takes this information and generates C code that uses the EmbedDB query API to execute the query. The developer then copies the code into their project for use.

The advantage is that the expensive parsing and optimization is performed offline. This allows much higher performance during query execution as the SQL is translated into C code and then compiled.

To perform pre-compilation, an external database engine is used to parse the query. SQLite would have been a good choice given that its source code is in C, but it does not compile the query into a logical query tree. The database selected was HyperSQL[2] that fully supports the SQL standard and parses the query into a logical query tree suitable for conversion to C code.

The converter systematically checks for parts of the query that it can support as well as those that it cannot support. There is currently support for WHERE clauses involving columns and arithmetic expressions as well as several common functions such as `floor()`, `abs()`, and `round()`. Aggregate expressions are also supported along with the use of GROUP BY. HAVING filters are handled by applying additional filtering after the aggregation is computed. In general, the SQL converter supports everything EmbedDB's query API supports, except for joins.

## 4 EXPERIMENTAL EVALUATION

EmbedDB was evaluated on multiple hardware platforms, listed in Table 1, that cover a range of capabilities and are representative of typical embedded sensor devices. The largest device, a Raspberry Pi, runs Linux and is capable of supporting SQLite. The other hardware is too small for SQLite. The devices using a SD card for storage support a file API and isolate the database from raw flash storage. The DataFlash storage is raw memory without a flash translation layer. Using raw memory has higher read performance but requires the database to manage all aspects of flash storage including page placement and erasing. The DataFlash memory is unique as it has a considerably higher read-to-write ratio compared to SD cards. Systems such as Antelope and LittleD were unable to be ported to the hardware for evaluation.

The storage performance for sequential and random writes was measured by writing a large file of 10,000 pages either sequentially or randomly. Random and sequential read performance was measured by reading 10,000 pages and inspecting each byte on the page. These metrics are the maximum possible I/O performance on the hardware. The high sequential read performance on the Raspberry Pi may be impacted by operating system buffering.

Multiple data sets were used for the experimentation as given in Table 2. These data sets include environmental monitoring data (**uwa**), smartwatch X/Y/Z

[2]https://hsqldb.org/

```
// Create EmbedDB state
embedDBState* state = (embedDBState*) malloc(sizeof(embedDBState));
state->keySize = 4;  state->dataSize = 12;

// Insert records
uint32_t key = 47; int32_t data[] = {23, 32, -90};
embedDBPut(state, (void*) &key, (void*) data);

// Query for single key
int success = embedDBGet(state, (void*) &key, (void*) data);

// Query for key and data range
embedDBIterator it;
uint32_t itKey; int32_t itData[3];
uint32_t minKey = 100, maxKey = 1000; int32_t minData=50;
it.minKey = &minKey;   it.maxKey = &maxKey;
it.minData = &minData; it.maxData = NULL;
embedDBInitIterator(state, &it);
while (embedDBNext(state, &it, (void*) &itKey, (void*) itData)) {
        /* Process record */
}
embedDBCloseIterator(&it);
```

Figure 3: Query Processing Example.

Table 1: Hardware Performance Characteristics.

| | | **Reads (KB/s)** | | **Writes (KB/s)** | | |
|---|---|---|---|---|---|---|
| **Device** | **Storage** | **Seq** | **Random** | **Seq** | **Random** | **R/W Ratio** |
| Raspberry Pi 3 Model B (4-core 1.2GHz, 1 GB RAM) | SD Card | 167075 | 36208 | 25600 | 25559 | 1.42 |
| ARM Cortex M0+ SAMD21 (48MHz, 32KB RAM) | SD Card | 483 | 414 | 435 | 326 | 1.27 |
| ARM Cortex M0+ SAMD21 (48MHz, 32KB RAM) | DataFlash | 495 | 491 | 36 | 37 | 13.63 |
| Arduino Due (32-bit 84 MHz, 96KB RAM) | SD Card | 980 | 1011 | 589 | 579 | 1.75 |
| Arduino Mega 2560 (8-bit 16MHz, 8KB RAM) | SD Card | 409 | 381 | 330 | 379 | 1.00 |

gyroscope data (**watch**), and ethylene concentrations over time (**ethylene**). The time series data points are 16 bytes consisting of a 4 byte integer timestamp and three 4 byte integers of sensor data. The experiments measure insert throughput, query performance, number of I/Os, and memory consumption. The metrics are the average of three runs.

SQLite was configured to use an `INTEGER PRIMARY KEY` for the timestamp value and a secondary index on the data value. If there were multiple data values in the timestamp record, only the data value attribute involved in queries was indexed. All inserts were performed in a single batch transaction for highest performance with the use of WAL logging and synchronous setting turned off.

## 4.1 Insert Performance

Insert performance was measured by inserting 100,000 records from the data sets. The performance is compared with the theoretical maximum performance of the device given its sequential write capability. The maximum performance is not achievable in practice as it does not consider any overheads related to CPU time or additional I/Os required for maintaining the data sets and associated indexes. However, it gives a metric for comparison in the absence of comparable systems capable of running on these devices. The maximum performance is independent of the data set and is shown as a black diamond in the graphs.

EmbedDB is compared with SQLite on the Raspberry Pi running Linux, which is the only test device capable of running SQLite. The results are in Figure 4 for the Raspberry Pi and Figure 5 for the other hard-



SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 2: Experimental Data Sets.

| Name | Points | Sensor Data |
|---|---|---|
| uwa (Zeinalipour-Yazti et al., 2005) | 500,000 | temp, humidity, wind, pressure |
| ethylene (Fonollosa et al., 2015) | 4,085,589 | ethylene concentration |
| watch (Stisen et al., 2015) | 2,865,713 | smartwatch X/Y/Z gyroscope |

ware platforms. The throughput is measured in inserts per second with higher numbers being preferred.

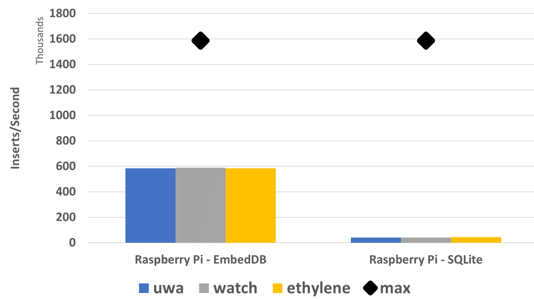


Figure 4: Insert Performance on Raspberry Pi.

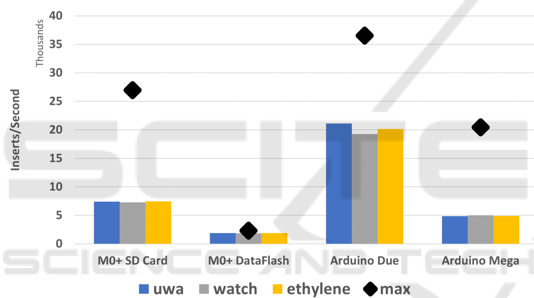


Figure 5: Inserts with EmbedDB on Smaller Devices.

There are two insights from these results. First, EmbedDB significantly outperforms SQLite for insertion performance. Even when SQLite is configured to batch all inserts, it is still about 10 times slower than EmbedDB. SQLite performance can be improved by a factor of two by removing an index on the data value, but that results in a major reduction for data queries. EmbedDB is highly efficient for inserts with no overhead in parsing or during execution. Although EmbedDB is not specifically designed for more powerful hardware, it has very good performance.

On the Arduino and custom hardware with lower memory between 8 to 32 KB RAM, EmbedDB's insert throughput is between 24% to 83% of the theoretical maximum write performance. SQLite achieves 3% of theoretical maximum performance on the Raspberry Pi while EmbedDB achieves 37%.

The difference between the actual performance and theoretical maximum performance depends on hardware characteristics. For slow write DataFlash, the system is I/O bound so EmbedDB is able to maximize its relative performance compared to the theoretical maximum. When using the faster SD card, CPU and other processing time is now a significant factor which reduces the relative percentage. On the Arduino Mega with a very slow processor, the CPU is the bottleneck and writing is slower.

## 4.2 Query Performance

Query performance is tested for three query types:

- Retrieving one record by key (timestamp)
- Retrieving a range of records by key (timestamp)
- Retrieving records that have data values in a given range

For all query experiments, 100,000 records were inserted, then the queries were executed.

### 4.2.1 Record Retrieval by Key (Timestamp)

Record retrieval by key was measured by performing 100,000 random key lookups on the data set created by inserting 100,000 records in the previous insert test. The results are in Figures 6 and 7. The theoretical maximum performance displayed on the charts is the expected throughput if the system performs one random I/O on the storage device and has no CPU overhead. This ideal case is not achievable.

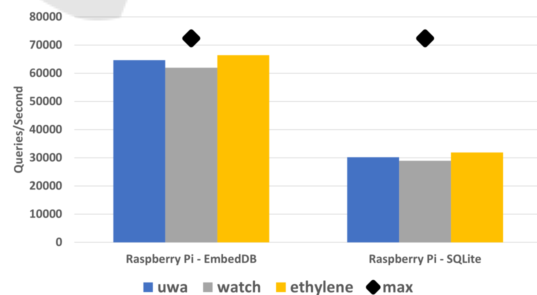


Figure 6: Record Retrieval by Key on Raspberry Pi.

EmbedDB has a factor of two speedup compared to SQLite achieving 86 to 92% of the theoretical maximum performance. There is a significant difference between the two systems on how the key lookup is performed. SQLite uses a B+-tree while EmbedDB uses a learned index (Ding et al., 2023). EmbedDB

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

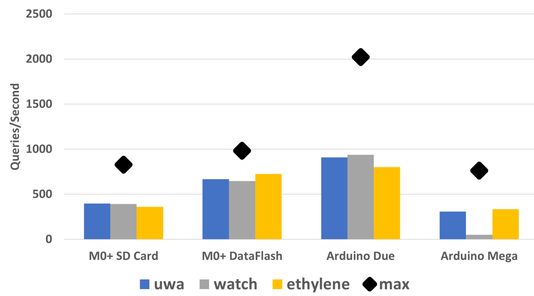


Figure 7: Record Retrieval by Key.

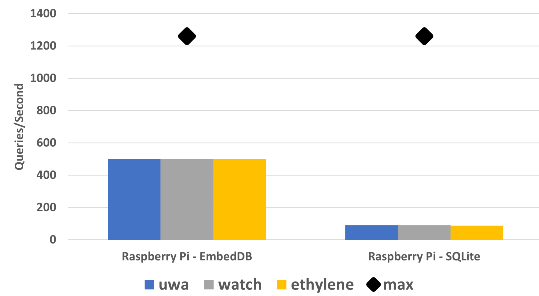


Figure 8: Key Range Retrieval on Raspberry Pi.

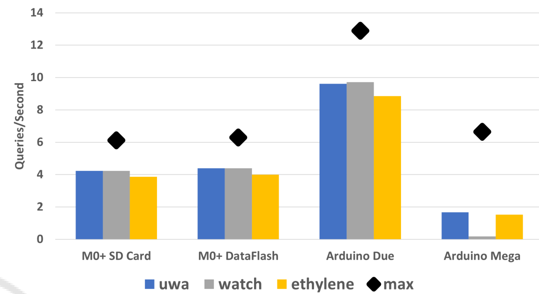


Figure 9: Key Range Retrieval.

uses 1.25 I/Os on average to retrieve a record. Its improved performance is also due to its low overhead query API that is efficient in returning the data record to the client application.

EmbedDB achieves between 40 to 70% of the maximum theoretical performance on the smaller hardware devices. In these systems, the slower CPU is a factor besides just performing the random I/O to retrieve the page containing the record. Once the page is retrieved, the system must search the page for the record, and extract it from the page to return to the client program. One outlier is the watch data set on the Arduino Mega. Due to the high variability in this data set, the space required for the learned index was over 2 KB, which was too large to maintain in the 8 KB RAM. EmbedDB adapted by using binary search in this case. The query performance is consistent and stable with 1% variance between runs. This is especially important for embedded and real-time applications where consistent performance is critical.

#### 4.2.2 Key Range Retrieval

Performance on retrieving a range of keys is important for timestamp queries requiring a window of records between a start and end time. Key range performance was measured by a query requesting 5% of the data set starting at a given key (timestamp). The performance of a key range query is impacted by how rapidly the system can find the starting key value using its index and then its scan performance to read the rest of the keys in the range. Since the keys are inserted in sorted order, sequential I/O should be performed once the starting key is found.

For comparison, the theoretical maximum is calculated as the time to scan 5% of the data set ignoring the cost to find the first key value and any CPU processing required. Due to the query requiring both sequential and random I/O, the average of the sequential and random I/O speeds were used for the read speed. As in previous experiments, the theoretical maximum is an unachievable baseline used only for comparison. The results are in Figures 8 and Figure 9.

EmbedDB excels in the key range retrieval task achieving 40% of the theoretical maximum performance with a five times speedup compared to SQLite. The efficiency of record scanning is the primary factor in performance for this test. SQLite has dynamic typing of columns resulting in overheads in record storage and data type conversion. EmbedDB is a key-value data store storing user-defined byte arrays. This enables optimized record storage in pages and minimal overhead in returning data to the client application. EmbedDB is also five times faster for scanning the entire data set.

Key range scan is highly efficient on all hardware platforms. Except for the Arduino Mega, performance is between 63 and 75% of theoretical maximum and is consistent across all data sets. The CPU-limited Arduino Mega has slower performance.

#### 4.2.3 Data Range Retrieval

For sensor applications, the time series data may measure environment information like temperature, humidity, air pressure, or sensor data such as X/Y/Z acceleration. These queries often search for outliers or points of interest in the time series, such as temperatures that are significantly different than normal.

Data range retrieval is evaluated by providing a selective range of data values to retrieve. The range is chosen to return about 0.01% of the data set. The selectivity determines the number of records retrieved and the performance of the approach.

Table 3: SQL Query Results.

| # | Query |
|---|---|
| 1 | **Min, max, avg daily temperatures**<br>`SELECT key/86400, min(temp), max(temp), avg(temp)`<br>`FROM uwa`<br>`GROUP BY key / 86400` |
| 2 | **Avg. temp on days with wind speed > 15 mph**<br>`SELECT key/86400, avg(temp), max(wind)`<br>`FROM uwa`<br>`GROUP BY key/86400`<br>`HAVING max(wind) > 15` |
| 3 | **Records with ethylene concentration > 0**<br>`SELECT COUNT(*)`<br>`FROM ethylene WHERE conc > 0` |
| 4 | **Compare temperatures at two weather stations in different years**<br>`SELECT u.key, s.temp as TempSEA, u.temp as TempUWA`<br>`FROM sea s JOIN (SELECT key + 473385600 as key, temp FROM uwa) u ON s.key=u.key`<br>`WHERE s.key >= 1420099200`<br>`AND s.key < 1451635200` |
| 5 | **Records with X motion above 500,000,000**<br>`SELECT key/706000 as Bucket, COUNT(*)`<br>`FROM watch`<br>`WHERE x > 500000000`<br>`GROUP BY key/706000` |

The benchmark is the expected time to complete a scan of the data set. Unlike the previous theoretical benchmarks, this is not a theoretical maximum that is unachievable. Instead, the database should be able to improve upon using a full table scan to answer the query by using indexing on the data values.

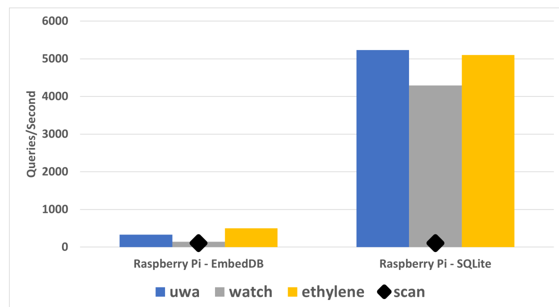


Figure 10: Data Range Query Performance on Raspberry Pi.

The data range results are in Figures 10 and 11. EmbedDB uses a bitmap index to reduce the number of pages read. The bitmap index allows it to only identify pages to read that potentially may have a data value in the range. The bitmap index nicely adapts to different data selectivities. When the data range covers most of the data set, most of the pages are read. When the data range covers only a small amount of the data set, there is a significant reduction in pages read. The data set distribution affects the data range query performance for EmbedDB. Using the bitmap index, EmbedDB reads between 6 to 20% of the data set, which results in a performance improvement of a factor between 3 and 15 compared to sequential scan.

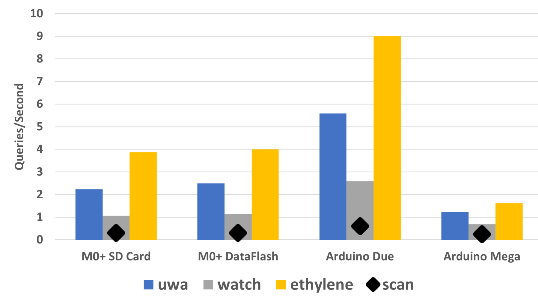


Figure 11: Data Range Query Performance.

Of the data sets, the **uwa** temperature data set is fairly regular as temperature follows a pattern. The **ethylene** data set is even more regular and slowly changing as it monitors a chemical reaction over time. The bitmap data filter is especially efficient in only reading pages that may contain the queried data values. The watch data set is highly variable as the X/Y/Z gyroscope data changes frequently in both sign and direction. This results in lower performance as more data pages must be scanned to answer the query.

When configured with a B+-tree secondary index on the data value, SQLite has superior performance compared to EmbedDB. The very high selectivity of 0.01% in the experiment favors the B+-tree index. As a larger percentage of records are returned, EmbedDB's bitmap index that performs sequential I/Os has higher performance than SQLite's B-tree index. This tradeoff of secondary indexes versus scans is well-known. Implementing a secondary B-tree index that performs significant random I/Os has performance issues for small embedded devices.

#### 4.2.4 Memory Usage

The memory used by EmbedDB was 2308 bytes: 200 bytes for state management, 20 bytes for file interface, 4 page buffers totaling 2048 bytes, and the key index of 40 bytes for the **uwa** data. Only the size of the key index is data set dependent. For **ethylene** it was 72 bytes, and **watch** was 2432 bytes.

### 4.3 SQL Query Performance

EmbedDB's advanced query API supports SQL operations such as filtering, grouping, and aggregation.

Figure 12 shows SQLite performance versus EmbedDB for the SQL queries in Table 3 where the query execution code was either developer produced or automatically generated by the translation tool.

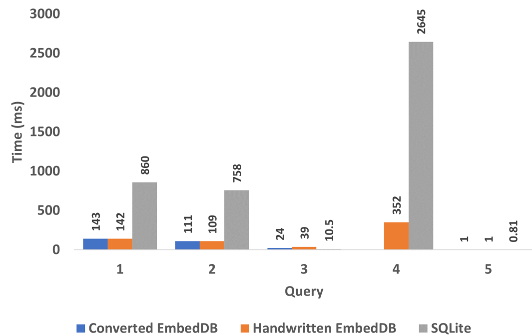


Figure 12: SQL Query Performance.

EmbedDB has faster scan performance than SQLite that translates into a speedup factor for SQL queries involving key filtering of about six times. SQLite outperforms for queries dominated by secondary data lookup with high selectivity (queries 3 and 5), but EmbedDB remains competitive while consuming less memory. There is no performance difference between developer created versus automatically generated SQL execution code. Query 4 with a join is not currently supported by the translator.

## 5 CONCLUSIONS AND FUTURE WORK

EmbedDB is a key-value, time series database supporting embedded and sensor devices, especially devices with hardware resources too small to execute other systems. Its database API consumes minimal resources and enables high performance data operations with limited coding by developers. Experimental results demonstrate its high performance on small devices, and competitive performance with SQLite for time series analysis on the Raspberry Pi.

Ongoing work is improving EmbedDB's performance with specific focus on adding data compression and further improving the efficiency of SQL query processing. EmbedDB is available as open source for researchers and developers at https://github.com/ubco-db/EmbedDB.

## REFERENCES

Anciaux, N., Bouganim, L., and Pucheral, P. (2003). Memory Requirements for Query Execution in Highly Constrained Devices. In *VLDB*, pages 694–705.

Bonnet, P., Gehrke, J., and Seshadri, P. (2001). Towards Sensor Database Systems. MDM '01, pages 3–14, London, UK, UK. Springer-Verlag.

Ding, D., Carvalho, I., and Lawrence, R. (2023). Using learned indexes to improve time series indexing performance on embedded sensor devices. In *SENSORNETS 2023*, pages 23–31. SCITEPRESS.

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., Ould-Khessal, N., and Lawrence, R. (2021). Efficient flash indexing for time series data on memory-constrained embedded sensor devices. In *SENSORNETS 2021*, pages 92–99. SCITEPRESS.

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

Gaffney, K. P., Prammer, M., Brasfield, L. C., Hipp, D. R., Kennedy, D. R., and Patel, J. M. (2022). SQLite: Past, Present, and Future. *VLDB*, 15(12):3535–3547.

Javed, F., Afzal, M. K., Sharif, M., and Kim, B.-S. (2018). Internet of Things (IoT) operating systems support, networking technologies, applications, and challenges: A comparative review. *IEEE Communications Surveys & Tutorials*, 20(3):2062–2100.

Madden, S. R., 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.

Ould-Khessal, N., Fazackerley, S., and Lawrence, R. (2021). B-tree Implementation for Memory-Constrained Embedded Systems. In *19th Int'l Conf on Embedded Systems, Cyber-physical Systems, and Applications (ESCS)*. CSREA Press.

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

Raspberry Pi Foundation (2023). Raspberry Pi. https://www.raspberrypi.org/.

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

Tsiftes, N. and Dunkels, A. (2011). A Database in Every Sensor. SenSys '11, pages 316–332, New York, NY, USA. ACM.

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. volume 4 of *FAST'05*, pages 31–44. USENIX.