Key-Value Store Implementations for Arduino Microcontrollers

Scott Fazackerley[†], Eric Huang, Graeme Douglas, Raffi Kudlac, Ramon Lawrence
Department of Computer Science
University of British Columbia, Kelowna, British Columbia, Canada

***Abstract*—The Internet of Things relies on connecting devices big and small to the Internet and facilitates their communication and management. This includes the smallest microcontrollers and embedded processors that perform simple but critical tasks. There are many applications where there is benefit to processing data locally on the device before sending it over the network. Since network communication is an order of magnitude more costly than local data processing, efficient techniques for storing, searching, and filtering data are useful. In this work, we present a library of implementations of key-value stores for use on Arduino devices. The goal is to make it easy for Arduino programmers to manipulate data without worrying about implementing data structures and query libraries. Key-value stores are conceptually simple for programmers to use. This paper describes the implementations and provides insights on their performance and trade-offs. The work has being released as open source to the Arduino community and is available at https://github.com/iondbproject/iondb.**

I. Introduction

As devices are connected to each other and continually process more data, there is a performance benefit to processing data as close to its source to reduce latency and communication costs. Data collection devices are often simple microcontrollers with very limited memory and CPU resources. Yet, even simple filtering and local storage can greatly improve device lifetime and energy usage while reducing network bandwidth usage.

Databases provide abstraction from storing and querying data and greatly improve programmer efficiency. Although there have been some efforts to create relational databases on microcontrollers such as LittleD [1], PicoDBMS [2], TinyDB [3], and Antelope [4], the device limitations restrict the features that can be supported and the performance obtainable.

Key-value stores encompass a class of data storage systems commonly refereed to as NoSQL systems due to a lack of a structured query language. Elements are referenced by a key value. Elements can take many forms such as documents, objects or discrete values. Key-values stores [5]–[10] are increasing popular due to their simple query interface. There are a variety of key-value algorithms for computers [11]–[13] with excellent performance. To our knowledge, there have been no implementations of key-value stores for embedded processor platforms like the Arduino. This work implements and benchmarks several key-value store implementations using a variety of data structures that can be deployed on an Arduino. The primary goal is to provide various key-value store implementations that Arduino developers can use in their own projects to simplify data management. A secondary goal is to provide some education on data structure implementations as many Arduino developers are students learning about programming and data structures. The library has the same API for all data structures and is easily extensible to new data structures and algorithm variations.

[†]Corresponding Author: scott.fazackerley@IEEE.org

The contributions of this work are:

- Four open source implementations of key-value stores for Arduino microcontrollers: two are memory-based and two have file-based persistence.
- A flexible framework for rapidly deploying and changing the key-value store's underlying storage structure and performance constraints.
- An experimental evaluation of the performance of the implementations and guidelines for programmers on how to select a suitable implementation for their data management use case.

The organization of this paper is as follows. Section 2 presents background information on embedded devices, the Arduino platform, and key-value stores. Section 3 provides an overview of the key-value store implementations, and Section 4 contains experimental results. The paper closes with future work and conclusions.

II. Background

Embedded devices are playing an ever increasing and important role in daily life. On a daily basis, humans interact with hundreds of devices which all generate data. Recent interest has focused on how this data can be harnessed and what underlying intelligence is contained within it. Due to the vast amount of data that is generated by devices, the ability to process data on device is beneficial as it reduces the amount of data that must be transferred off the device.

A new paradigm called the Internet of Things (IoT) has been gaining ground and momentum. Traditional models of centrally stored data will tax existing systems. Visions proposed by IoT pioneers such as Cisco [14], view the IoT as a decentralized system where data is shared directly between devices, driving the need for local storage and processing. Analysts [15] forecast that by the end of this decade, the IoT will range between 30 billion and 50 billion devices. Infrastructure and storage is a key challenge.

The drive toward the Internet of Things is enabled through devices such as the Arduino [16] computing platform. What started off as a small project to support a group of students has

grown into a world-wide phenomena. The Arduino platform has simplified development of projects allowing users without significant computing and hardware skills to develop and build Internet enabled devices that can share information. This movement has seen the infiltration of Arduino devices into research [17], creative endeavors and wearable technologies [18].

One of the key requirements of the IoT is the ability for devices to connect and share data wirelessly [19]. Devices have the ability to collect vast amounts of information. In practice, only a small subset is of interest and usually pertains to a specific transitory or periodic event in the sampling space. Pottie and Kaiser [20] have demonstrated that the amount of energy to transmit 1Kbyte over a link of 100 meters is equivalent to a processor executing 3 million instructions at 100 MIPS/Watt, which strongly supports the benefit of efficient sorting and processing of local data.

The Arduino platform has a large user base and well documented and supported hardware and software. The Arduino is based on the AVR and ARM microprocessors from Atmel. Key performance attributes are presented in Table I. When designing for Arduino, users must constantly be aware of the limited primary (SRAM) and secondary storage available. The Arduino does not have an effective data storage layer. Current storage is limited to flat files without a reasonable method of managing data.

TABLE I. COMPARISON OF ARDUINO MODEL SPECIFICATIONS[1]

Model	Processor	EEPROM [KB]	SRAM [KB]	Flash [KB]
Uno	ATmega328	1	2	32
Nano	ATmega168/ATmega328	0.512/1	1/2	16/32
Mega 2560	ATmega256	4	8	256
Leonardo	ATmega32u4	1	2.5	32
Due	AT91SAM3X8E	-	96	512

Database systems simplify data management by isolating users via an API. Even the smallest relational databases such as SQLLite[2] use too much memory to be viable on an Arduino or embedded device. For example, SQLite requires at least 200 KB of code space. Previous work on sensor networks such as COUGAR [21] and TinyDB [3] implemented a relational query processor on sensor nodes which were controlled/programmed from a server computer. This is not suitable for semi-autonomous programs running on embedded systems. There has been work on implementing relational databases on smart cards in the PicoDBMS [2] project. The most feature-rich relational databases for embedded devices are Antelope [4] and LittleD [1]. Both research systems implement a relational database with support for SQL. Experimental results in [1] show that although a relational database can be implemented on an embedded processor, the parsing of SQL takes up considerable code space and is a limiting factor. Further, queries on embedded devices are often quite simple so the overhead of SQL and a relational database may be too much. Non-relational alternatives include BerkeleyDB[3] and UnQLite[4], but are unsuitable due to memory requirements.

[1]http://arduino.cc/en/Products.Compare
[2]http://www.sqlite.org/
[3]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html
[4]http://www.unqlite.org/

Key-value stores [5]–[10] have become increasing popular due to their simple query interface and high performance. To the best of our knowledge, there is no key-value store available for the Arduino. Unlike the relation model which involves potentially complex syntax, a key-value store provides a simple interface making it suitable for resource constrained devices.

This work introduces IonDB, a generic key-value store API for Arduino and embedded processors that is extensible for many different implementations. This paper provides several reference implementations and benchmarks their performance. The goal is to simplify query processing using key-value stores compared to Arduino developers having to develop data management functionality from scratch.

III. KEY-VALUE STORE API

The IonDB key-value store for the Arduino platform is a flexible and dynamic framework. It offers an interface that allows the Arduino user to store and query key-value data on device while choosing structures that address the runtime and storage requirements of the application. The API allows for dynamic binding of different storage structures.

The core of the IonDB key-value store is composed of a dictionary_handler structure and dictionary API. The key-value store API is designed to be dynamically bound to compatible data structures. The underlying data structure handler is required to include an initialization function that is responsible for binding implementation of the specific data structure functions to the handler's function pointers. This allows different implementations to be bound to different instances at the same time. It allows developers the flexibility to customize their underlying storage structure while maintaining consistency in the interface. The user is required to define an initialization function for each type of underlying data store. Listing 1 shows an example of the required API mapping for a file based implementation.

Listing 1. API Function pointer binding
```
void ffdict_init(
   dictionary_handler_t *handler)
{
   handler->insert   = ffdict_insert;
   handler->update   = ffdict_update;
   handler->get      = ffdict_query;
   handler->find     = ffdict_find;
   handler->remove   = ffdict_delete;
   handler->create_dictionary
                     = ffdict_create_dictionary;
   handler->delete_dictionary
                     = ffdict_delete_dictionary;
}
```

Before a specific instance of a dictionary can be used with a data structure, the `init` function must be called to bind the required functions to the API. The API is composed of the following functions:

A. Dictionary Creation

The function `create_dictionary` creates an instance of the dictionary using a specific data structure and returns the status of the creation. It binds the underlying data structure through the `dictionary_handler` to an instance of the key-value dictionary. It is also responsible for registering key

size, key type and value size and the size of the dictionary (in terms of the number of records). The dictionary size is required for data structures that are of fixed capacity such as an open address hashmap. The type of key is also indicated (signed or unsigned numeric, or string) so the correct equality comparator can be bound to the instance. A generalized set of endian agnostic comparators are provided as a service to the user, eliminating the need for data structure specific implementations.The general type signature for API functions is shown in Listing 2.

Listing 2. API Function Signature

```
status_t
dictionary_get(
   dictionary_t      *dictionary,
   ion_key_t         key,
   ion_value_t       value
);
```

For all functions, the first parameter is the pointer to the dictionary instance to be operated on, with the following parameters being `ion_key_t` and `ion_value_t` as required. The types `ion_key_t` and `ion_value_t` are base type pointers allowing for implementation specific definition, allowing for flexible cross platform use.

The IonDB API also provides complete functions for constructing predicates and comparing keys of arbitrary size. The functions can be used directly by the caller without having to use additional supporting code at the data structure and handler level.

B. Insert and Update Operations

The `dictionary_insert` function allows the inserting of a value and associated key into an instance of a dictionary. The behavior of the dictionary can be defined to allow for unique entries or to support duplicate entries. This behavior depends on the operation of the underlying data structure and is controllable at run time. The function will return the status of the insert.

Existing values can be updated using the `dictionary_update` function. The behavior of the update operation depends on the underlying implementation in terms of the number of duplicate entries to be updated. The recommended behavior is that all values with key matching the update key are updated in the operation. The API includes a `write_concern` parameter that is used to control the behavior when dealing with duplicate entries. If the `write_concern` is set as `wc_insert_unique`, the insert operation will function as a strict insert. When `write_concern` is set as `cw_update`, it will allow the insert function to be transformed to an `upsert` type operation. If the key entry exits in the store the `upsert` operation will update the associated value otherwise, the key-value pair will be inserted into the store. This functionality allows the developer of the underlying data structure to reuse the insert function as an update operator.

C. Query Operations

The function `dictionary_get` will return a value that is associated with the key. By convention, the caller will pre-allocate memory for the return `ion_value_t`. The function returns the status of whether a key was found. If the data structure has the ability to store duplicate keys, the function will return the first entry that matches the key.

The `dictionary_find` function allows queries over a range of key values using a predicate. Predicates allow for strict equality, inclusive range query or a complex predicate statement. Through the use of unions, a single predicate struct allows for multiple different statement types as well as supporting runtime binding of a destroy function pointer. This eliminates the need for the caller to be concerned of underlying memory allocation across multiple predicate types. The caller is required to provide an instance of an unallocated `ion_cursor_t` to the function. The callee is responsible for memory allocation and function pointer binding.

The `dictionary_find` method dynamically binds the iterator function as `next()` and a destroy function to the cursor instance. This allows the cursor implementation to take advantage of specific attributes present at the data structure level.

When calling the `next()` function for a given data structure, the caller is responsible for pre-allocation of the return value `ion_value_t`. The status of the iteration is returned through the `status_t` return type. This allows the caller to determine if additional values are associated with the cursor instance. The IonDB cursor is forward-only and like other data stores, results are produced in an unstable, unsorted order.

Unlike traditional databases that may offer consistency in the resultset, IonDB does not have the ability to offer consistency guarantees. As results are generated from a query, the iterator traverses and materializes results from the data structure on a record-by-record basis. The results associated with a given cursor instance may change due to a write operation on the data structure during the existence of the cursor. If this occurs, the return status from the next call indicates that the results of the query may no longer be consistent. This allows for the caller to decide how to proceed with the existing cursor.

D. Delete Operations

The `dictionary_remove` function will delete a key-value pair based on a strict match using the compare function that is associated with the dictionary instance. For data structures that allow duplicate records, the delete function will delete all instances of the associated key-value pair.

The `dictionary_delete` function deletes the underlying data structure instance associated with the dictionary instances as well as freeing any other resources held by the current dictionary instance. This results is the complete destruction and removal of the given dictionary instance and all key value data.

E. Key-Value Store Implementations

Four different underlying data structures have been implemented for use with the IonDB key-value store API. All implementations are written in C, allowing for portability to other devices. The underlying structures implemented are a skip-list, hashmap, flat file and file based hashmap. The first two are strictly in-memory structures for use on devices with

no external storage capacity. They offer different advantages in terms of runtime and space complexity. Consider the family of devices presented in Table I. There is a significant difference in available resources in terms of on-device storage, SRAM and code space. While some devices such as the Due can maintain a large number of records in SRAM, the Uno is considerably more limited and may require the use of a data structure that is deterministic in SRAM requirements.

For devices that require significantly more storage, two disk based data structures are presented that utilize the Arduino SD shield and FAT16 SD card library. Due to the modular design of IonDB, the implementation does not have hard requirements on the underlying storage medium. Although C does not directly support the overloading of functions, the API provides preprocessing directives that allow the developer to redefine I/O functions from `stdio.h`. This allows for a high degree of customization for storage interfaces, allowing for other types of storage media to be considered in the future.

1) Skip List: The skip list [22] is a dynamically sized, in-memory structure designed for the storage of records when the average speed is crucial. The implementation is simpler than that of a balanced tree, as well as offering better performance [22]. The size of the skip list in terms of the number of entries is bounded by the available memory of each device (see Table I). It provides an average insert of $O(\log n)$, where n is the number of records in the structure. As shown in Figure 1, the skip list is an ordered linked list augmented with an array of forward pointers. Every node in the skip list has some height h, and a node at height i will have some fixed probability p that the node also appears at height $i + 1$. In this implementation, the probability can be configured by the end user as necessary.

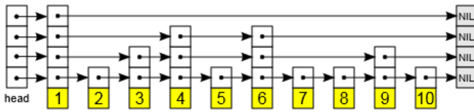


Fig. 1. Skip List Organization

To insert, the head node is set as the current node. Starting at the highest height node, the structure traverses forward in the linked list. The current node is only moved forward if its key is less than or equal to the node being examined. It is not moved forward if the value of the node is nil. If moving forward is not possible due to these conditions, the cursor is moved down one height level and the search process is repeated. When the cursor cannot move forward or downward any further, the insert position has been located. Since each node is of varying height, traversal across the upper heights allows the cursor to “skip” over nodes known to be smaller than the current key. All other operations are similar in behavior to the insert and only vary in terms of the specific operation they perform once the required cursor position in the skip list has been located.

Equality and range query implementations are straight forward due to the ordered property of the skip list. Equality is executed by traversing the list until the first node with a matched key is located. The cursor then traverses linearly across the bottom of the ordered linked list until the key match fails. A range query follows a similar implementation where it searches the skip list for the first node that satisfies the lower bound and upper bound defined by the predicate for the query. Once found, the cursor continues in a linear traversal until the key of the current node no longer satisfies the predicate.

The skip list offers good performance in terms of searching for key matches and supporting duplicate keys without introducing significant overhead in runtime costs. It has significant overhead due to the number of pointers that are required to be maintained for each node in the list to support the skipping behavior. This further constrains the maximum number of elements that can be stored in device SRAM.

2) Open Address Hash Map: The open address hash map is a statically sized, in-memory structure designed to have predictable memory consumption. The hash map is based on an associative array where keys are mapped to specific locations in the storage array based on a hashing function. The implementation of the in-memory hash map uses open addressing where the in-memory array stores the key, value, and element status byte in a series of contiguous hash buckets (Figure 2). This results in very low memory overhead as the size of each entry in the hash map is

$$\begin{aligned} \text{bucket space size} &= sizeof(ion_key) \\ &+ \; sizeof(ion_value) \\ &+ \; sizeof(status). \end{aligned} \tag{1}$$

Unlike other linked structures such as the skip list, the hash map does not need to maintain pointers to other elements in the data structure.

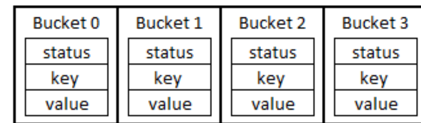


Fig. 2. Hash Map Bucket Organization

To insert a key-value pair into the data structure, the correct bucket needs to be determined for the given key. This is accomplished using the hashing function associated with the data structure implementation. The implementation supports dynamic runtime binding of hashing functions, allowing the user to supply their own to adjust the performance of data structure and avoid clustering of data. A simple modulo based hash function is provided with the implementation. Once the hashed-to bucket has been determined for a given entry, the status of the bucket is determined. On initialization all buckets are set to empty. If the hashed-to bucket is empty, then the key-value pair is entered into the bucket and the status set to occupied. If the slot is occupied, a probe sequence is initiated where successive buckets are scanned looking for a bucket that has a status of empty or deleted. The probing will continue through the array and wrap back to the first bucket to continue probing. Probing continues until an unoccupied or deleted bucket is located at which time the contents will be updated and the status set to occupied. If the probing continues until it returns to the hashed-to bucket without finding an unoccupied slot, the data structure will return as being at maximum capacity.

To locate a value, update or delete a key-value pair, the same operations are performed as with an insert to determine the hashed-to location, except that during the probing operation the key of each bucket is examined for a match. If found, the desired operation is completed. The probing continues until the cursor probe location returns to the original hashed-to location.

The performance of the hash map is dependent largely on the load factor, that being the ratio of the number of locations occupied in the data structure. The higher the load factor, the larger the number of buckets that must be probed to find the key. If a perfect hash can be obtained where one and only value maps to a single location, operations are $O(1)$. Due to the open addressing scheme and the uncertainty in the distribution of hashed keys, operations are $O(n)$ where n is the number of buckets in the hash map. This behavior is further exacerbated by delete operations. When searching for a key, as any key that is not immediately located in the hashed-to location could be in any other location as a result of open addressing all bucket entries may need to be examined in the worst case.

Equality and range query operations are subject to the same performance constraints as with insert type operations due to the unordered nature of the structure. With equality operations as soon as a match is found, it can return the value. With the range query, the entire structure must be scanned to locate a key in the worst case.

While the run-time complexity of the open address hash map is less desirable than other structures, its simple in memory structure, low memory overhead and deterministic memory footprint makes it a desirable for use on memory constrained devices.

3) Flat File: One of the most significant limitations for the Arduino family of devices is the extremely limited amount of SRAM that is available to store data. Sampled data can exceed the amount of available SRAM. Though the Arduino libraries[5] offers the ability to store data on SD cards, the interface is limited in terms of what can be done with files. The user must handle how data is stored and located in the file. The flat file implementation utilizes the SD file API to create a FAT16 disk based storage structure freeing the user from having to manage files directly.

The flat file stores records in no general order in a flat file. The record structure is similar to that of the record defined in Equation (1) offering low overhead with a single status byte. The data structure supports both duplicate keys and unique key only operations. For operations where duplicate keys are allowed, records are directly appended to the end of file resulting in fast inserts. If the data structure supports unique keys only, the file is traversed in a linear fashion from the head to the tail looking for a key match. If a matching key is found, the value will be updated if `write_concern` is set as `wc_update` otherwise it will indicate that duplicate keys are not allowed. If the end of the file is reached without finding a match, the record will be appended to the end of the file.

The find, update and delete operations are similar, where the file is scanned in a linear fashion from the head looking for a match. The existence of duplicate or unique keys further impacts scanning operations in the flat file. If the flat file is allowed to store duplicate keys, the scan operation will start at the head of the file, scanning forward checking for a key match. It will continue scanning until it reaches the end of the file. If the structure only stores unique entries, it will stop scanning upon finding a match. In the case of the delete operation the status byte will be set to tombstone the location upon a key match, rendering the record deleted.

Due to the unordered nature of the records in the flat file, both the equality and range cursors operate in the same fashion. In both cases, the cursor starts scanning from the head of the file checking key match conditions. In the case of equality, if the structure supports unique keys only it will return a match. In the case of duplicate keys, both the equality and range cursors must traverse the entire file checking for matches.

The search of a key has runtime complexity in the order of $O(n)$ where n is the number of entries in the flat file. The performance is dominated by SD I/O operations. The implementation is considerably simpler than other disk based structures and offers a significantly sized key-value data store that is limited in size only by the size of available storage.

4) File Based Open Address Hash Map: The file based open address hash map is a statically sized, disk based structure designed to have predictable disk utilization. The implementation of the structure is similar to the in-memory implementation in all cases. The difference is that instead of creating the associative array in memory, the array is created on disk in a single file. This allows the device to have a considerably larger implementation than with the in-memory hash map. Like with the flat file structure the performance is dominated by SD I/O costs.

IV. Experimental Results

The performance of the four different structures were tested on the Arduino Uno and Arduino Mega2560. Both devices run at a clock speed of 8 Mhz. Additional details can be found in Table I. A testing framework was constructed that allowed unit and profiling tests to be run for comparison on the Arduino devices. Timing was conducted using the internal free running clock counters in the target processor. Test suites were executed multiple times with the end results being averaged. In memory structures were run on the Arduino Mega2560 and file based structures on the Arduino Uno.

Precomputed test records could not be used in testing as they would occupy unnecessary code space. All tests were done with unique keys as some of the underlying data structures do not support duplicate keys. While the avr-gcc implementation of random number generator can be used to generate random keys, duplication is possible leading to inconsistent profiling. To over come this, a Galois Linear Feedback Shift Register was implemented in the framework which generates a pseudo random sequence not containing duplicate values within a cycle.

Tests were conducted for insertions, query, and delete operations over a series of unique keys for all implementations. The results for in-memory data structures are shown in Figure 3, and the results for file-based data structures are shown in Figure 4. For memory based implementations, query and delete

[5] http://arduino.cc/en/Reference/Libraries

operations were done on a store that initially contained 200 unique keys. For file based implementations, query and delete operations were done on a store that initially contained 100 unique keys. The graphs show the operation time per record in milliseconds based on the current size (bytes) of the data store.

In analysis of inserts between the skip list (*SL*) and open address hash map (*OAHM*) implementation, the OAHM is faster, but both grow linearly at the same rate in terms of the number of records inserted. For query operations, as the number of records (bytes) requested or stored in the data store increased, the SL maintains consistent performance per record but the OAHM decreases linearly in performance due to the open addressing scheme.

In analysis of inserts between the file based open address hash map (*FHM*) and flat file (*FF*) implementation, the FHM is faster. This is due to the constrained nature of the file and the lack of repeated scanning that is required for the FF implementation. For query operations, as the number of records (bytes) requested or stored in the data store increased, both structures decrease in performance due to increase in file I/O. The FF decreases at a significantly greater rate due to the repeated full file scans. For delete operations, both data structures offer consistent performance as the size of the structure grows. The FF has worse performance due to the repeated scanning of the complete file required by the implementation.

Table II shows the overhead required for each data structure. The in-memory overhead is once per instance while the record overhead is per record inserted into the structure. Table III presents performance options to consider when choosing an implementation. For memory constrained devices, the open address hash offers deterministic memory usage with reasonable performance. If memory utilization is less of a concern, the skip list offers strong performance but with the cost of higher memory overhead. The flat file offers the user the ability to store data persistently, in an unbounded fashion but at the cost of degrading performance while the file base open address hash map offers better performance as well as a upper bounded on file size.

TABLE II. DATA STRUCTURE MEMORY OVERHEAD

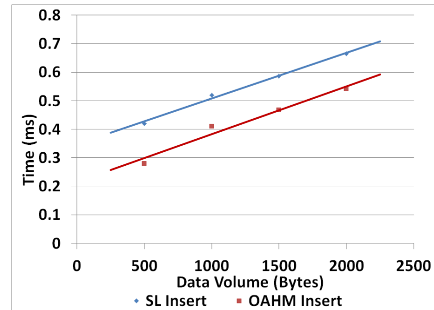
	In Memory Overhead (Bytes)	Record Overhead (Bytes)
Skip List	110	28
Open Address Hash Map	19	1
Flat File	78	1
File Based Open Address Hash Map	13	1

V. CONCLUSIONS

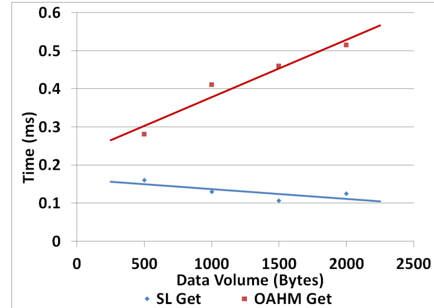
This work presents the first key-value store API for use with the Arduino family of devices. The IonDB key-value store offers a collection of data storage structures that can be easily implemented by a user without having to understand the underlying storage infrastructure. This work presents two in-memory and two disk based implementations with varied storage and runtime performance. For highly memory constrained devices

TABLE III. PERFORMANCE CONSIDERATIONS

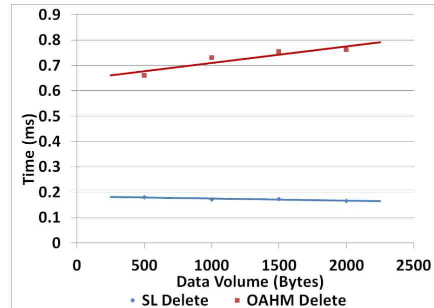
	Memory Utilization	Runtime	Strengths	Weakness
Skip List	Fair	Excellent	Excellent performance for all operations	High memory overhead, lack of persistence
Open Address Hash Map	Excellent	Good	Good performance, Low and deterministic memory overhead	Performance degrades as load factor increases, lack of persistence
Flat File	Good	Fair	Simple implementation, can grow in reasonably unbounded fashion, persistent	Insert and query performance degrades with number of records
File Based Open Address Hash Map	Excellent	Fair	Consistent performance on inserts. Good performance for other operations, persistent	Performance degrades as load factor increases but not as quickly as flat file



(a) Insert Operations

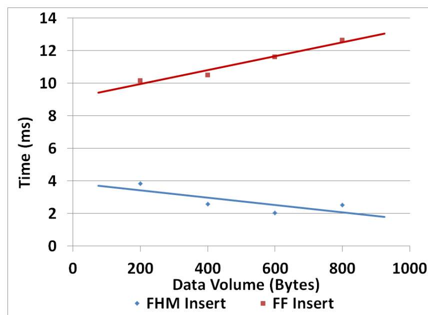


(b) Query Operations

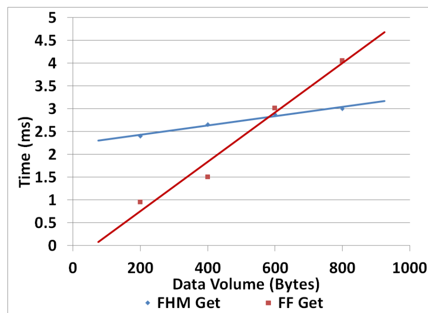


(c) Delete Operations

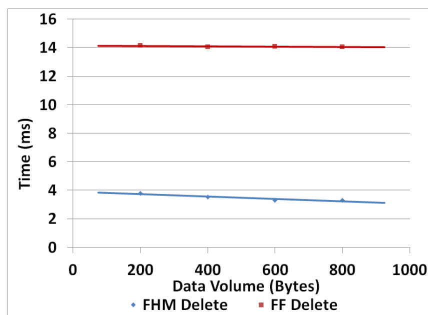
Fig. 3. Timing for Skip List (SL) and Open Address Hash Map (OAHM) In-Memory Data Structures on the Arduino Mega2560



(a) Insert Operations



(b) Query Operations



(c) Delete Operations

Fig. 4. Timing for File Based Open Address Hash Map (FHM) and Flat FIle (FF) Data Structures on the Arduino Uno

such as the Arduino Uno, the file based storage structures offer large scale storage outside of main memory while devices with larger SRAM can enjoy the performance of in-memory structures.

The IonDB API open source framework provides infrastructure for implementing and improving data storage structures while maintaining a consistent caller code space for embedded devices. This provides significant opportunity for users to understand and profile different data structures for use in key-value stores across any C compatible machine. It is available at https://github.com/iondbproject/iondb. Future work will increase the number of available data structures in addition to improving the performance of the current implementation.

References

[1] G. Douglas and R. Lawrence, "LittleD: A SQL Database for Sensor Nodes and Embedded Applications," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 827–832.

[2] N. Anciaux, L. Bouganim, and P. Pucheral, "Memory Requirements for Query Execution in Highly Constrained Devices," ser. VLDB '03. VLDB Endowment, 2003, pp. 694–705.

[3] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, Mar. 2005.

[4] N. Tsiftes and A. Dunkels, "A Database in Every Sensor," ser. SenSys '11. New York, NY, USA: ACM, 2011, pp. 316–332.

[5] Z. Wei-ping, L. Ming-xin, and C. Huan, "Using MongoDB to implement textbook management system instead of MySQL," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on* , May 2011, pp. 303–305.

[6] G. Lu, Y. J. Nam, and D. Du, "BloomStore: Bloom-Filter Based Memory-Efficient Key-Value Store for Indexing of Data Deduplication on Flash," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, April 2012, pp. 1–11.

[7] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-value Store," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1414–1425, Sep. 2010.

[8] Y. Kang, R. Pitchumani, T. Marlette, and E. L. Miller, "Muninn: A Versioning Flash Key-Value Store Using an Object-based Storage Model," in *Proc. of International Conference on Systems and Storage*, ser. SYSTOR 2014. New York, NY, USA: ACM, 2014, pp. 13:1–13:11.

[9] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 1–13.

[10] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 25–36.

[11] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sep. 1977.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, Jun. 2012.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[14] D. Evans, "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything," Cisco Internet Business Solutions Group, Tech. Rep., 2001.

[15] C. Witchalls, "The Internet of Things Business Index: A Quiet Revolution Gathers Pace," The Economist Intelligence Unit Limited, Tech. Rep., 2013.

[16] C. Severance, "Massimo Banzi: Building Arduino," *Computer*, vol. 47, no. 1, pp. 11–12, Jan 2014.

[17] M. Thalheimer, "A Low-Cost Electronic Tensiometer System for Continuous Monitoring of Soil Water Potential," *Journal of Agricultural Engineering*, vol. 44, no. 3, p. e16, 2013.

[18] L. Buechley, M. Eisenberg, J. Catchen, and A. Crockett, "The LilyPad Arduino: Using Computational Textiles to Investigate Engagement, Aesthetics, and Diversity in Computer Science Education," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '08. New York, NY, USA: ACM, 2008, pp. 423–432.

[19] D. Giusto, A. Iera, G. Morabito, and L. Atzori, Eds., *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications*. Springer, 2010.

[20] G. J. Pottie and W. J. Kaiser, "Wireless Integrated Network Sensors," *Commun. ACM*, vol. 43, no. 5, pp. 51–58, May 2000.

[21] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards Sensor Database Systems," ser. MDM '01. London, UK, UK: Springer-Verlag, 2001, pp. 3–14.

[22] W. Pugh, "Skip lists: A Probabilistic Alternative to Balanced Trees," in *Algorithms and Data Structures*, ser. Lecture Notes in Computer Science, F. Dehne, J.-R. Sack, and N. Santoro, Eds. Springer Berlin Heidelberg, 1989, vol. 382, pp. 437–449.