Adapting Linear Hashing for Flash Memory Resource-Constrained Embedded Devices

by

Andrew Feltham

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

B.SC. COMPUTER SCIENCE HONOURS

in

The Irving K. Barber School of Arts and Sciences

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 2019

Abstract

Linear hashing is a key-value data structure with constant time operations that is widely used for indexing in database systems. The research goal was to implement this data structure on embedded devices such as Arduino which have limited memory. Storing data on embedded devices is increasingly important for environmental sensor and Internet of Things applications. Flash memory persistent storage, which has unique properties with read and write times, presents an additional challenge to construct an efficient implementation. Several implementations were created and tested. This talk will explain the different linear hash implementations that were created and present the benchmarks collected.

Table of Contents

List of Figures

Acknowledgements

With thanks to Dr. Ramon Lawrence who guided and supported me through this adventure.

Chapter 1

Introduction

Linear hashing dates back to work done by Litwin [Lit80] and later expanded by Larson [Lar82, Lar85]. Linear hashing is an expandable hash table on storage that provides constant time operations. Although B+-trees are generally favored for database workloads as they also provide ordered access, linear hashing is implemented in many relational database systems and has benefits for certain use cases. In the embedded domain, linear hashing is interesting as it may allow for even better performance and less resource usage than B+-trees.

In this work, linear hashing is adapted and optimized for flash-based, memory-constrained embedded devices and shown to work for devices with as little as 8 KB of memory. Optimizations include implementing a linked list of overflow buckets in a backwards chaining fashion to avoid writes, trading off writes for reads due to asymmetric performance of flash memory, and minimizing the memory consumed so that most operations require only one memory buffer and at most two memory buffers are required for a split during insert.

1.1 Embedded Devices

Embedded devices are small, low powered, computer systems. Many embedded devices exist today for use within the open source community. One of the most popular is the Arduino project which produces multiple embedded computers with varying capabilities. This work used an Arduino 2560 Mega as a benchmarking and testing device.

This Arduino device has 8KB of SRAM and a 16Mhz cpu. Developing with so little available RAM produces a challenge for implementing efficient data structures.

1.2 Flash Memory

Flash memory is commonly used in embedded devices as a long term storage. An ethernet shield with a microSD slot was used for the Arduino 2560 Mega used in this work.

SD cards are rated by their sequential read or write speed and have varying performance depending on their class. In general read speeds are better than write speeds and sequential access is faster than random. These memory characteristics guided the development into using less writes and potentially favouring sequential over random access.

1.3 Embedded Databases

There have been several efforts to construct database libraries and software tools for these embedded devices starting with the sensor-database networks such as TinyDB [MFHH05] and COUGAR [BGS01] to database software installed and executing on the device such as Antelope [TD11], PicoDBMS [ABP03], LittleD [DL14], and IonDB [FHD+15]. There have also been data structures and algorithms specifically developed for flash-memory including [GT05, LZYK+06]. This work attempts to improve an existing implementation of Linear Hashing built into the IonDB project.

1.4 Motivations

There is a renewed focus on data processing on devices with limited capabilities as applications such as sensor-based monitoring grow in deployments. The Internet of Things [LYZ+17] relies on these devices for data collection and filtering, and it is widely known that there are performance and energy benefits to processing data on the edge (where it is collected) rather than sending it over the network for later processing. Manipulating data on these edge devices represents similar challenges to the early days of computing with limited resources and supporting software. Implementing a low memory and efficient hash table for these devices is beneficial and an interesting experiment.

1.5 Contributions

This project utilized the IonDB [FHD+15] framework to implement the linear hash table. The honour thesis by Spencer Macbeth which in-

vestigated linear hashing on embedded devices was expanded on in this work [Mac17]

Chapter 2

Background

First introduced by Litwin in 1980 [Lit80], the linear hash data structure is a dynamically-resizable hash table which maintains constant-time complexity for hash table operations. A search generally takes about one access, and the space utilization may be up to 90%. This performance is superior to B+-trees for key-based lookup operations. Linear hashing does not require an index to lookup bucket locations on storage if the buckets are allocated continuously on storage or allocated in fixed size regions. Computing the address of a record is done by using the output of the hash function computed on the key to identify the appropriate region (if multiple) and bucket within the region. Thus, the memory consumed is minimal and consists of information on the current number of buckets and next bucket to split.

2.1 Hashing

The key feature of a linear hash table is that expansion occurs one index at a time as needed instead of doubling the table size as in a regular hash table. This is made possible by the hashing function and splitting mechanism. Two hashing functions are used to calculated the table index. Equation 2.1 and Equation 2.2. These equations require the hash table properties N as the initial number of buckets, and L the number of times the hash table has doubled in size. Algorithm 2 is used to calculated the table index for a given key hash. *nextSplit* in this algorithm is the next bucket that will be split. This algorithm finds the last number of bits in the key hash that is a valid hash table index. Every time the linear hash table doubles in size additional bits in the hash are used to find the table index.

$$H_0(H) = H \bmod (N \times 2^L) \tag{2.1}$$

$$H_1(H) = H \bmod (N \times 2^{L+1}) \tag{2.2}$$

Overflows in a bucket are handled by adding a new bucket to a linked list of buckets. Overflow buckets are chained together as needed.

Algorithm 1: Hashes a key to a table index

```
if H_0(H) < nextSplit then
    index ← H_1(H)
else
    index ← H0(H)
end
```

An example table is shown in Figure 2.1. This figure shows hashed keys inserted into a table with an initial size of 4. This table demonstrates key hashing and overflow chaining.

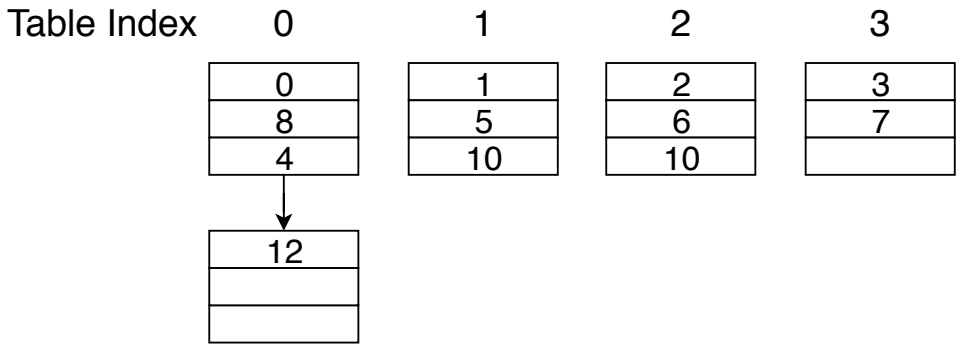

Figure 2.1: Example linear hash table with hash keys

2.2 Split

The hash table is dynamically resized when the storage utilization (load factor) increases beyond a set amount. At that point, a new bucket is added to the end of the hash file and records are divided between the new bucket and the current bucket to split in the table. It is this predefined, ordered splitting of buckets that is the main contribution of linear hashing.

The load factor is calculated using Equation 2.3. After every insert operation this load factor is re-calculated and compared against the maximum load factor. If it is exceeded then a split operation is triggered.

$$load = \frac{totalRecords}{size * recordsPerBucket} \tag{2.3}$$

During a split operation the table size is expanded by one and a new bucket is created. The bucket referenced by the *nextSplit* is loaded and split into the newly added bucket. Due to the way the hashing works, all records that should belong in the new table index are inside the buckets at the *nextSplit* index. Records are moved if the H_0 and H_1 functions have different values. Once the split has completed, if the table size has now doubled, the *nextSplit* is reset to the first index otherwise it is incremented by one.

An example split of a split is shown in Figure 2.2. In this example a new bucket was added to the end of the table and records from table index 0 were moved to this new bucket.

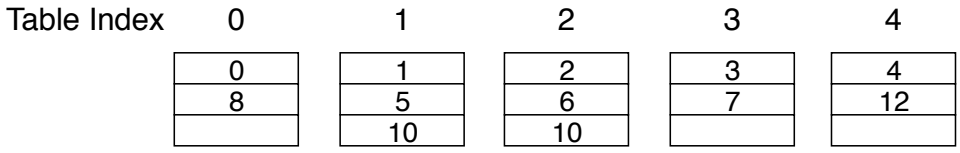

Figure 2.2: Example linear hash table with hash keys after a split

2.3 Variations

Linear hashing was extended and generalized by Larson [Lar82] using partial expansions. It was shown that performance can be increased if doubling of the file size is done in a series of partial expansions with two generally being a good number. Search performance is increased at the slight trade-off of additional algorithm complexity and the need for buffering and splitting $k + 1$ buckets in memory where k is the number of partial expansions. Further work [Lar85] allowed for the primary buckets and overflow buckets to use the same storage file by reserving pre-defined overflow pages at regular intervals in the data file. This work also added the ability to have multiple overflow chains from a single primary bucket by utilizing several hash functions to determine the correct overflow chain.

Variations of linear hashing optimized for flash memory use the idea of log buffering to increase performance. The Self-Adaptive Linear Hash [YJYZ16] buffers logs of successive operations before flushing the result to storage. This often decreases the total number of read and write operations and allows for some random writes to be performed sequentially. Self-Adaptive Linear Hash also adds higher levels of organization to achieve more

coarse-grained writes to improve the bandwidth. Unfortunately, the extra memory consumed is impractical for embedded devices.

Chapter 3

Implementation

To fully explore optimizations of linear hash tables on the embedded architecture and flash memory, several implementations of a linear hash table were constructed, benchmarked, and compared. This chapter describes the implementations with their optimizations and intended uses.

3.1 Common Implementation

Since the goal of this project was experimenting with different possibilities for optimization of the linear hash table on flash memory, a basic framework of a linear hash table was created and modified for the different implementations described later in this chapter. Due to the shared core functionality of a linear hash table, many of the structures, functions, and logic were identical. This section describes the common structures used in each implementation.

3.1.1 Linear Hash Table

The linear hash table structure was used to keep track of variable values required to perform the operations.

3.1.2 Records

Records are stored in buckets in binary format. Each record contained a key and a value as sequential values. Separators were not used between the key and value as the keys and values are required to be constant sizes which is set during the linear hash table creation. The linear hash table precalculated the record size during initialization.

3.1.3 Buckets

A linear hash table is built around buckets which contain records and metadata describing the bucket and its contents. The same bucket structure is used for top level buckets in each hash table index and overflow buckets

that are linked together as a linked list. Each bucket can contain up to a maximum number of records which was calculated using equation 3.1.

$$recordsPerBucket = floor(\frac{blockSize - bucketHeaderSize}{recordSize}) \quad (3.1)$$

The bucket object was implemented using a C struct object. This struct contains the hash table index it belongs to, the number of records stored in the bucket, a pointer to the next bucket in the chain, and the remainder of the space was the record data. To optimize file reading and writing performance, each bucket was set to be 512 bytes in size to align to the file system's block boundaries. The header used 12 bytes total leaving 500 bytes for the bucket data. Buckets are loaded from the files using buffers, modified in memory and written out to the file as entire blocks.

Overflow buckets are handled by linking buckets together as a linked list. The `overflow block` field in the header points to the logical block in the overflow location (implementation dependent) where the next bucket in the chain exists. The `UINT32_MAX` value was used to indicate there the was no overflow bucket for a particular bucket. This value set a hard limit on the number of buckets that could be created.

3.2 Buffers

The buffer structure was used to load a bucket block from a file and keep track of values related to that buffer. Two buffers were allocated for each implementation as a minimum of two are required during a split operation to read a bucket and move records into another bucket. However, most functions required only one buffer. These buffers were allocated when the linear hash table implementation was created so that it can fail early if allocation failed. This is the main memory impact of the implementations. Each buffer had to store both the 512 bytes of data plus the additional buffer flags, resulting in 1042 bytes of memory overhead.

A dirty flag was used to save some unnecessary writes during splits, deletes, and updates. The buffer contents were written and read only as entire blocks of data.

3.3 Hashing

Hashing keys to index values is the key feature of the linear hash table. Several key hashing functions were tested to find a function that offered rea-

sonable distribution and performance for the embedded environment. Using the work done by Fritter et al., the SDBM function was found to offer well rounded performance on the Arduino hardware [FOKFL18]. This function was implemented as the default key hashing function. A function pointer was added on the linear hash table structure to allow easy replacement of the hashing function.

A key was hashed to the hash table index with Algorithm 2. This function was used at the beginning of every operation and was not modified for any of the three implementations.

Note that this function is dependent on the linear hash table values *nextSplit* and *initialSize*. These values are modified as the hash table grows during the split operation.

Algorithm 2: Hashes a key to a table index

Input: The key to hash
Output: The table index where that key should be found
Function `getIndex(`key`)`
  $hash \leftarrow SDBM(key)$
  $index \leftarrow H_0(hash)$
  if $index < nextSplit$ **then**
    $index \leftarrow H_1(hash)$
  end
  return $index$
Function H_0`(`$hash$`)`
  return $hash\&(initialSize - 1)$
Function H_1`(`$hash$`)`
  return $hash\&(2 * initialSize - 1)$

3.4 Hash Table Operations

3.4.1 Get

The get or find hash table operation is shown in Algorithm 3. This operation is just a linear scan through all the buckets in the bucket chain found at the index for the key.

Algorithm 3: Get Operation

```
Input: The key to find
Output: The value found or an error code
index ← getIndex(key)
bucket ← loadTopBucket(index)
terminal ← false
while ¬terminal do
    foreach record in bucket.records do
        if record.key = key then
            return record.value
        end
    end
    if bucket has overflow then
        bucket ← loadOverflowBucket(bucket.overflow)
    else
        terminal ← true
    end
end
return NotFound
```

Performance

The find operation requires exactly 0 writes as it does not perform any modifications to the hash table. The operation takes minimum O(1) and maximum O(N) reads where N is the number of buckets in the bucket chain at the given index.

3.4.2 Delete

Delete operations are a linear scan through the buckets in an index chain and matching records are removed. Since each bucket is loaded into memory, records are manipulated in place. This operation is outlined in Algorithm 4. During deletion records are shifted up in order to maintain a continuous block of records. This saves time during insertion as the insertion spot is immediately at the bottom of the currently used area. The algorithm accomplishes shifting with one pass through the bucket by keeping track of an insert and a read pointer. As the records are scanned through, the read pointer is incremented but the insert pointer is not incremented if a record should be deleted. If the read and insert pointer do not match after a record

is read, then the read record is moved to the insert pointer and the insert pointer is incremented.

In order to remove a bucket, the previous bucket in the linked list must be reloaded from the file and then modified to point to the bucket pointed at by the removed bucket. This process requires an extra read and write. In order to save writes and memory space, empty buckets were not removed from the overflow chains and records were not shifted between buckets. This can result in partially empty or empty buckets remaining in the bucket chain.

Performance

Deletion requires a linear scan through all the records in the bucket chain of length N which requires N reads. The writes required range between 0 and N depending on how many buckets were modified.

3.4.3 Split

Splitting was implemented with Algorithm 5. This algorithm ensures that the new bucket chain is continuous with entirely full buckets. A split is triggered when the load of the linear hash table is reached after an insert. This load is configurable on the linear hash table structure. Similar to deleting, empty gaps in the splitting chain are filled, empty buckets are not removed, and partially full buckets are not filled.

This function is the main reason two buffers are required. One buffer to load the splitting bucket and a second buffer for the new bucket.

Performance

Splitting requires reading every bucket in the bucket chain for an index (N) and writing every bucket in the chain as well as all newly created buckets. In the worst case where every record must be moved this takes 2N writes.

3.4.4 Insert

In general the insert functionality loaded the top level bucket and potentially further buckets as needed until an empty spot was found. If required a new overflow bucket was added. After an insert a split could potentially be performed. This operation was determined to be the operation where optimizations and experiments could be made. Therefore, each implementation modified this function and the algorithms are documented further in this chapter.

Algorithm 4: Delete Operation

```
Input: The key to delete
Output: The number of records deleted
deleted ← 0
index ← getIndex(key)
bucket ← loadTopBucket(index)
terminal ← false
while ¬terminal do
    insert, read ← firstRecordPointer
    count ← bucket.recordCount
    for i ← 0 to count do
        record ← recordAtReadPointer
        if record.key = key then
            bucket.recordCount− = 1
            deleted+ = 1
        else
            if insert ≠ read then
                copy record at read to insert
            end
            insert += recordSize
        end
        read += recordSize
    end
    write out bucket
    if bucket has overflow then
        bucket ← loadOverflowBucket(bucket.overflow)
    else
        terminal ← true
    end
end
return deleted
```

Algorithm 5: Split Operation

```
Input: The key to delete
Output: The number of records deleted
deleted ← 0
index ← getIndex(nextSplit)
splittingBucket ← loadTopBucket(index)
newBucket ← createNewBucket()
completed ← false
while ¬completed do
    insert, read ← splittingBucket.firstRecordPointer
    count ← bucket.recordCount
    for i ← 0 to count do
        record ← recordAtReadPointer
        hash ← SDBM(record.key)
        if H0(hash) ≠ H1(hash) then
            insert the record into the new bucket
            write out the newBucket if full and create a new overflow
            splittingBucket.recordCount −= 1
        else
            if insert ≠ read then
                copy record at read to insert
            end
            insert+ = recordSize
        end
        read+ = recordSize
    end
    write out splittingBucket
    if splittingBucket has overflow then
        splittingBucket ←
         loadOverflowBucket(splittingBucket.overflow)
    else
        completed ← true
    end
end
write out new bucket
currentSize++
if currentSize == 2 × initalSize then
    initialSize ← 2 × initalSize
    nextSplit ← 0
else
    nextSplit++
end
```

Once an insert has been made, the load is checked using Equation 2.3, compared to the maximum load set on the table. If the load is larger then the maximum load, a table split is made.

3.5 File Implementation

The file based implementation was built as a simple base implementation of a linear hash table. The goal of this implementation was to provide a performance comparison between a standard linear hash table and further attempts at modifications.

This implementation requires two files, data and overflow, to store the linear hash table contents. A third file was used to store the hash table state for simplicity.

The data file was used to contain top level buckets in the linear hash table. Each file block contained a bucket that matched the table index for that bucket. As the table was expanded new buckets were added to the end of the file.

Overflow buckets were added to the overflow file. When an overflow block was created, it was always appended to the end of the overflow file and then the previous bucket was linked to the logical file index for that file block.

3.5.1 Get

The get operation functions identically as in Algorithm 3. The only change required is to load the bucket from different files during $loadTopBucket$ and $loadOverflowBucket$.

3.5.2 Delete

The delete operation functions identically as in Algorithm 4. The only modification required is to load the bucket from different files during $loadTopBucket$ and $loadOverflowBucket$.

3.5.3 Split

The split operation functions as in Algorithm 5. The buckets are tracked if they are a top level or an overflow bucket in order to write it out to the correct file.

3.5.4 Insert

Inserting into the file based implementation by finding the index for the key, iterating through the buckets in the bucket chain until a bucket with space is found. If all bucket are full then a new bucket is added to the end of the chain.

The file performance for this function is best case O(1) reads and O(1) writes in the case where the top level block is not full.

In the worse case, where a new bucket must be added, this is n reads where n is the number of buckets in the current bucket chain and 2 writes in order to insert the new bucket and update the previous bucket overflow pointer.

3.6 Bucket Map

The bucket map implementation was built around the goal of reducing the number of writes and reads required for an insert and removed the extra file needed by the file based implementation.

The two data files that were required in the file based implementation were merged by always inserting buckets to the end of the file. In order to map the hash table index to a top level bucket an in-memory expandable array was added to the hash table that maps the index to a file block.

3.6.1 Insert

Insertion can now be done in exactly one read and write by inserting an overflow bucket to the top of the bucket chain instead of the bottom and only reading the top bucket before creating an overflow. This process is shown in Algorithm 7.

The downside of only reading the top level bucket before expanding is that partially empty buckets further in the chain will never be filled. This can impact the performance of the split, get and delete operations.

3.6.2 Get

Get functions identically as in Algorithm 3. The only difference is that the file location is retrieved from the bucket map before reading the bucket from the file.

Algorithm 6: Inserts a key and value into the hash table

```
Input: The key and value to insert
Output: Error or success code
index ← getIndex(key)
bucket ← loadBucketFromDataFile(index)
completed ← false
while ¬completed do
    if bucket is not full then
        insert record into the bucket
        write out bucket
        completed ← true
    else
        if bucket has overflow then
            bucket ← loadOverflow(bucket.overflow)
        else
            bucket.overflow ← nextOverflowBlock
            write out bucket
            bucket ← initalizeNewBucket()
            insert record into the bucket
            write out bucket
            completed ← true
        end
    end
end
numRecords += 1
if currentLoad > load then
    split()
end
```

Algorithm 7: BucketMap Insert Operation

Input: The key and value to insert
Output: The error or success code
$index \leftarrow getIndex(key)$
$fileIndex \leftarrow bucketMap[index]$
$bucket \leftarrow loadBucket(fileIndex)$
if *bucket is full* **then**
    $bucket \leftarrow initalizeNewBucket()$
    $bucket.overflow \leftarrow fileIndex$
    insert record into bucket
    $writeBucket(nextFileIndex)$
    $bucketMap[index] \leftarrow nextFileIndex$
    $nextFileIndex \mathrel{+}= 1$
else
    insert record into the bucket
    $writeBucket(fileIndex)$
end
$numRecords \mathrel{+}= 1$
if $currentLoad > load$ **then**
    $split()$
end

3.6.3 Delete

Delete functions identically as in Algorithm 4. The only difference is that the file location for the top level bucket is read from the bucket map.

3.6.4 Split

The split algorithm was modified to load the splitting bucket using the file location in the bucket map as well as updating the bucket map for the new bucket and its overflows. The new bucket chain is created in the same way as in the insert algorithm.

3.7 Serial Writing

An implementation was created with the goal of taking advantage of the flash memory performance for serial writes. The hash table was modified to always write changed and new buckets to the end of the file. This necessitated using the bucket map implementation in order to always know where the top level buckets were located.

3.7.1 Insert

Insertion was a simple modification of Algorithm 7 which simply wrote out the changed bucket to the next available file location and updated the bucket map.

3.7.2 Split

In order for splitting to work with overflow chains, the splitting bucket chain has to be reversed and rebuilt during the split. This was accomplished by reading the bucket chain top down and then updating both the bucket map and overflow pointers as further blocks were read and written out.

Chapter 4

Results

Benchmarks were performed on all tables by inserting random 2-byte keys and random 2-byte values. The linear hash table was configured with the SDBM hashing function and a load of 85%.

4.1 Block Statistics

Writing to flash memory file blocks was tested with three different SD cards. 10000 file blocks were written as random or sequential writes. A file of 10000 blocks was benchmarked with sequential and random reads.

These benchmarks are shown in Figure 4.1 and demonstrate the large variability of different SD cards and the general expected performance. In general it was found that read operations are faster then writes and sequential writes are faster then random writes. Sequential reads took an average of 47% of the time it takes for a sequential write. A random read takes an average of 64% of the time of a random write.

4.2 Insert

Insertion was benchmarked by recording the time to insert records into a hash table. These statistics are an average of five runs and include the time for splitting.

Figure 4.2 shows the time per insert for a range of inserts. It is obvious from this graph the the serial writing implementation did not perform as expected. This trend continued for all benchmarks and will be left out in later figures.

Figure 4.3 shows the same graph with the non-overwrite removed. It shows that the bucket map is faster as expected until around 70,000 records have been inserted. After this cut off the file based implementation performs better. This performance difference is likely a result of the time used to expand the bucket map array as well as additional time needed during a split due to increased bucket map chains.

Figure 4.1: SD Card benchmarks

Figure 4.4 and Figure 4.5 show the block reads and writes for the insert operation benchmarks. As expected the bucket map implementation is less then the file implementation and is around 1 read and write per insert operation.

4.3 Get

Get operations were benchmarked by getting random keys from a table of varying size. The number of gets was made as half the current table size.

Figure 4.6 shows that the bucket map implementation was variable and consistently slower then the file based implementation. This is expected as a result of larger partially filled bucket chains.

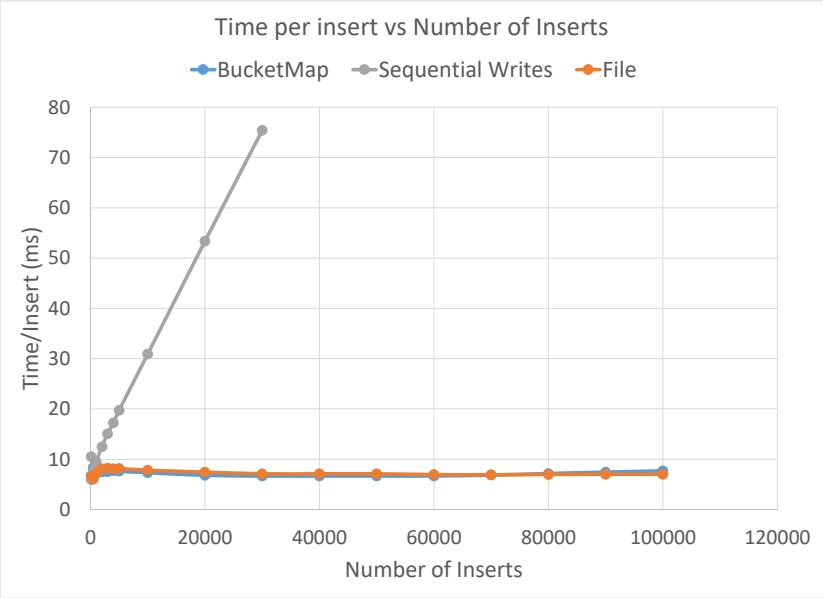

Figure 4.2: Time per insert

4.4 Delete

Delete operations were benchmarked by deleting random keys from tables of different sizes. The number of deletions was half the number of records in the table. However due to the random nature of the data some deletions failed to delete any records.

Figure 4.7 shows the benchmarked times for these delete operations. Bucket map was consistently slower then the file based implementation. Similar to the get statistics this is likely the result of longer bucket chains with partially filled buckets.

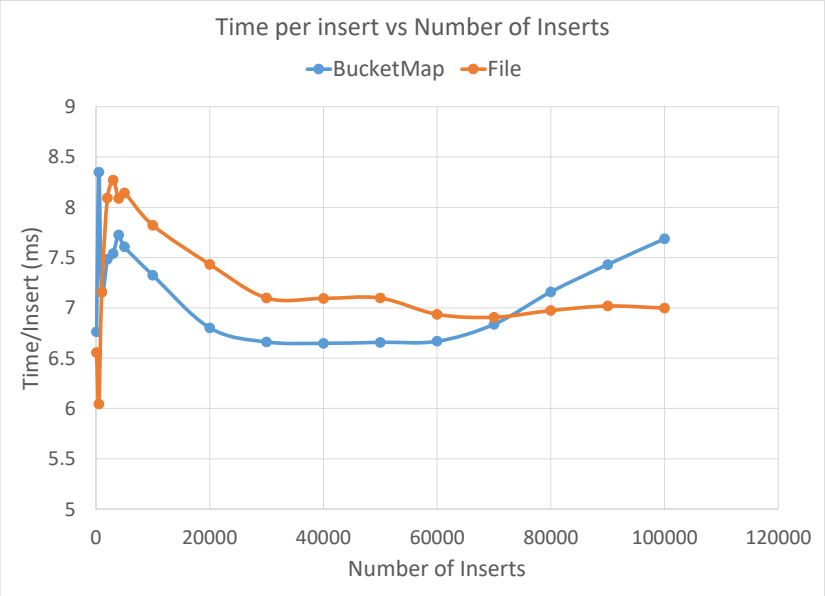

Figure 4.3: Time per insert

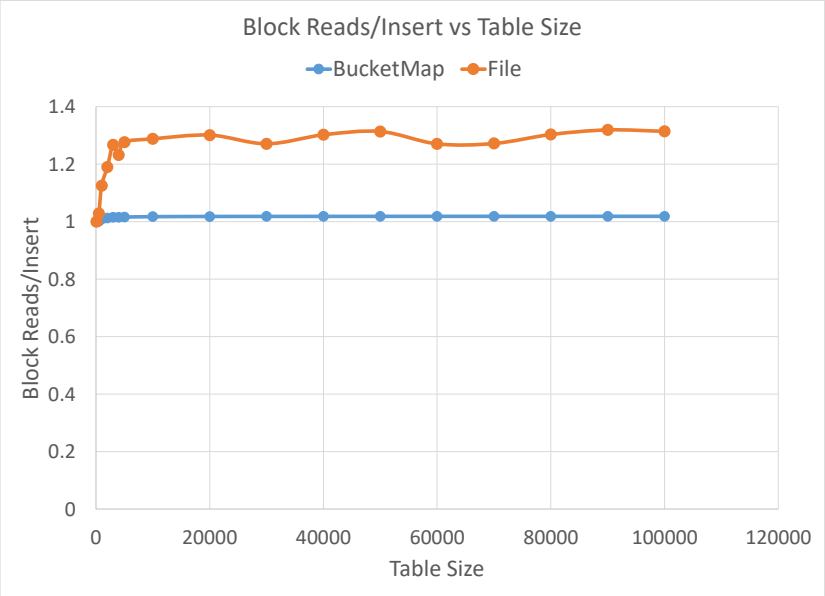

Figure 4.4: Block reads per insert

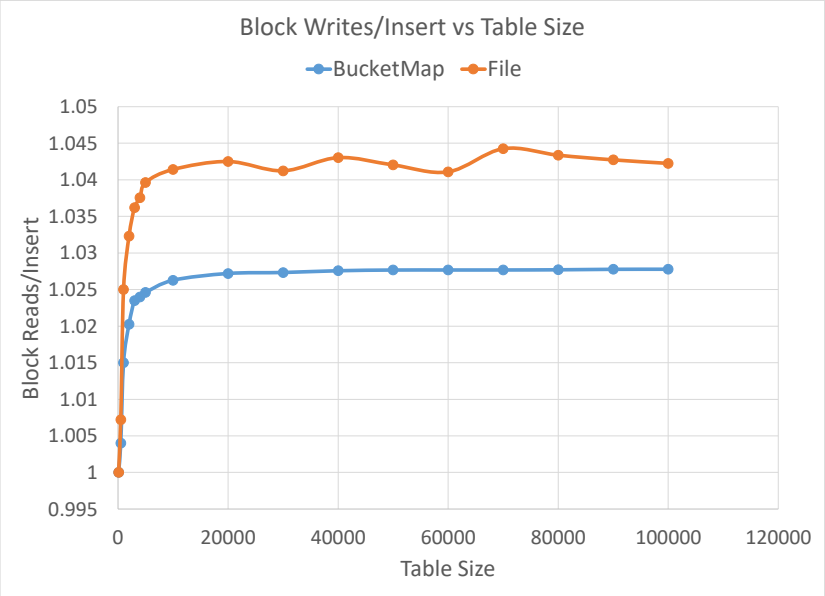

Figure 4.5: Block writes per insert

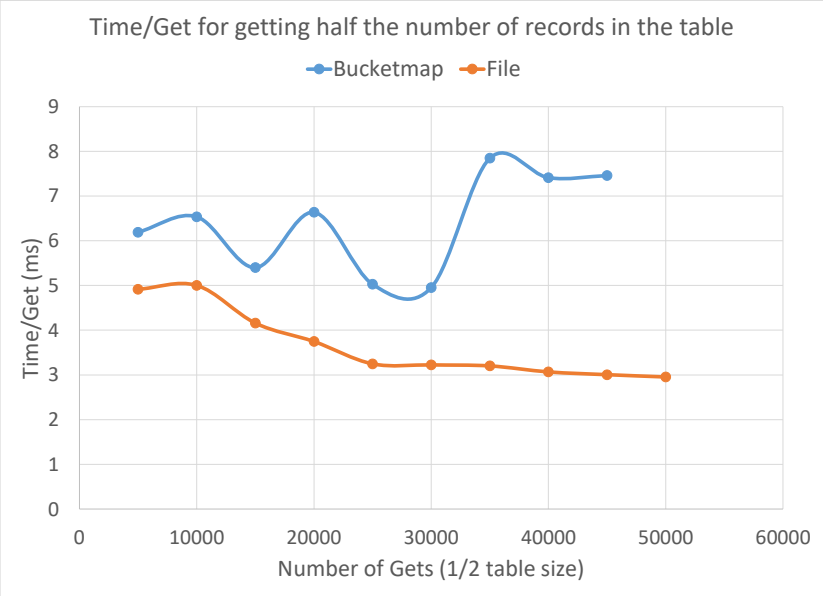

Figure 4.6: Time per get operation

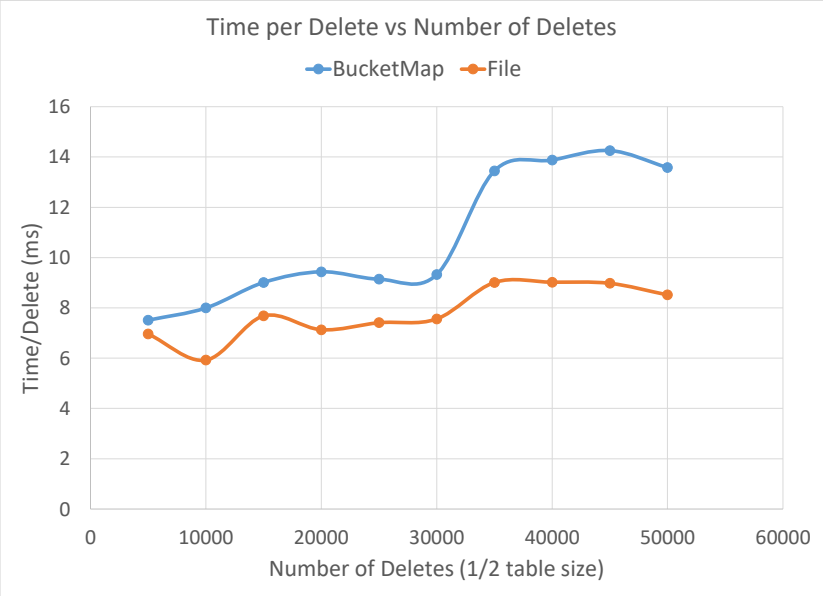

Figure 4.7: Time per delete operation

Chapter 5

Conclusion

The performance of three implementations of a linear hash table were compared. For table sizes under 70,000 the bucket map implementation has superior performance during insert operations. However, the bucket map implementation performed slower than the standard file implementation during get and delete operations. This is due to the increased bucket chains with partially filled buckets caused by the modifications to reduce the number of writes during insertion. The bucket map implementation also required larger amounts of memory as the table size grows in order to index all the top level blocks in the table.

The sequential writing implementation did not work as expected and resulted in a massive increase in time for all operations. We believe this is a result of increased seek times in the file system as a result of larger file size. Research into random reads with the FAT file system in an embedded environment has shown that many reads are required to seek to random locations as the file size grows [PFL16].

In conclusion, the bucket map implementation is efficient and has faster inserts compared to the standard implementation for the case of smaller table sizes. However, for large table sizes the file based implementation performed better.

5.1 Future Work

Future work on linear hash tables for embedded devices should investigate reducing the number of buffers required for the split operations, and investigate and improve the performance of the bucket map implementation. The sequential write implementation should be investigated to determine the cause of the massive slow down and be tested with direct file access. Additional linear hash table techniques such as partial expansions and could be explored for implementation.

Exploration into raw SD card access, ignoring a file system entirely, has potential to bypass the problems found with with the FAT file system.

Bibliography

[ABP03] Nicolas Anciaux, Luc Bouganim, and Philippe Pucheral. Memory Requirements for Query Execution in Highly Constrained Devices. VLDB '03, pages 694–705. VLDB Endowment, 2003. → pages 2

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In Kian-Lee Tan, Michael J. Franklin, and John Chi-Shing Lui, editors, *Mobile Data Management*, pages 3–14, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. → pages 2

[DL14] Graeme Douglas and Ramon Lawrence. LittleD: A SQL Database for Sensor Nodes and Embedded Applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 827–832, New York, NY, USA, 2014. ACM. → pages 2

[FHD+15] Scott Fazackerley, Eric Huang, Graeme Douglas, Raffi Kudlac, and Ramon Lawrence. Key-value store implementations for arduino microcontrollers. In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, may 2015. → pages 2

[FOKFL18] Matthew Fritter, Nadir Ould-Khessal, Scott Fazackerley, and Ramon Lawrence. Experimental evaluation of hash function performance on embedded devices. In *2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE)*. IEEE, may 2018. → pages 10

[GT05] Eran Gal and Sivan Toledo. Algorithms and Data Sructures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005. → pages 2

[Lar82] Per-Åke Larson. Performance analysis of linear hashing with

partial expansions. *ACM Transactions on Database Systems*, 7(4):566–587, dec 1982. → pages 1, 6

[Lar85] Per-Åke Larson. Linear hashing with overflow-handling by linear probing. *ACM Transactions on Database Systems*, 10(1):75–89, mar 1985. → pages 1, 6

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 212–223. VLDB Endowment, 1980. → pages 1, 4

[LYZ+17] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal*, 4(5):1125–1142, oct 2017. → pages 2

[LZYK+06] Song Lin, Demetrios Zeinalipour-Yazti, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Efficient Indexing Data Structures for Flash-Based Sensor Devices. *Trans. Storage*, 2(4):468–503, November 2006. → pages 2

[Mac17] Spencer Donald James MacBeth. Linear hashing for flash memory on resource-constrained microprocessors. 2017. → pages 3

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, mar 2005. → pages 2

[PFL16] W. Penson, S. Fazackerley, and R. Lawrence. Tefs: A flash file system for use on memory constrained devices. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–5, May 2016. → pages 28

[TD11] Nicolas Tsiftes and Adam Dunkels. A Database in Every Sensor. SenSys '11, pages 316–332, New York, NY, USA, 2011. ACM. → pages 2

[YJYZ16] C. Yang, P. Jin, L. Yue, and D. Zhang. Self-adaptive linear hashing for solid state drives. In *ICDE*, pages 433–444, May 2016. → pages 6