Adaptive extendible hash maps

CS265 final research report

Leonhard Spiegelberg
Institute for Applied Computational Science
52 Oxford Street
Cambridge, MA
spiegelberg@g.harvard.edu

Victor Lei
Institute for Applied Computational Science
52 Oxford Street
Cambridge, MA
vlei@g.harvard.edu

ABSTRACT

In this project, we aim to investigate the Extendible Hash Map (EHM) data structure and try to improve it by relaxing certain fixed parameters to assess the feasibility of creating a dynamic, adaptive EHM that has better performance than a standard EHM. A standard EHM can be seen as a combination of a binary prefix tree (or trie) that is flattened into an array and used to address buckets of fixed size. Usually, each bucket uses the same data structure to access the elements within a bucket. In this project, we seek to relax the fixed size assumption of the buckets, and change the internal bucket data structure such that each bucket can have a different size and internal data structure. Through several experiments, we investigate the performance impact of changing bucket size for workloads consisting of different read-write ratios, and assess the performance of three different bucket data structures as bucket size and read-write ratio change. We find that there is significant variation in performance across a variety of workloads as bucket size and internal data structure are varied, suggesting there is potential for a dynamic, adaptive EHM to outperform an optimized standard EHM.

Keywords

Data Structures; Extendible Hash Maps; Adaptive Data Structures

1. INTRODUCTION

Extendible Hash Maps (EHMs) are a commonly-used data structure in file systems and database systems. Their flexibility in expanding to accommodate large amounts of data, and their good performance characteristics, make them a versatile data structure. Typically, EHMs are designed with a set of fixed design parameters, including a fixed bucket size (often a set number of elements) and a fixed internal bucket structure (often a simple array.) We present evidence that through the use of a hybrid design that varies bucket size and internal bucket structure for each individual bucket, performance of EHMs can be improved over a standard, static EHM. Through several experiments, we first demonstrate that the performance of a standard EHM can depend upon the read-write ratio of a workload and the bucket size, and that this relationship is non-trivial. Then, we explore the effect of also varying the internal bucket structure between a simple array, a sorted array that allows for a binary search, and a B+ tree. We find that there is significant variation in performance across a variety of workloads as bucket size and internal data structure are varied, suggesting there is potential for a dynamic, adaptive EHM to outperform an optimized standard EHM.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. BACKGROUND

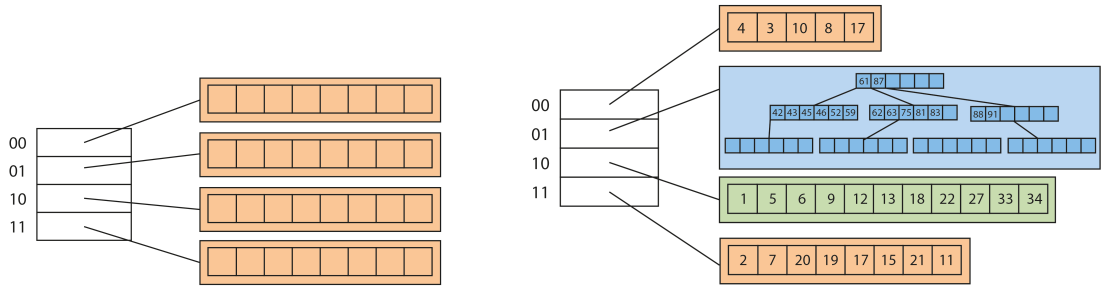
Traditional Hash Tables as a data structure provide an average complexity of $\mathcal{O}(1)$ with a worst case of $\mathcal{O}(n)$ for lookup, store and delete operations, assuming a collision-free hash function.

However, in reality there are several disadvantages associated with hash tables: First, hash tables need to have a fixed size and are thus a static data structure. If the size chosen is too small, many collisions will happen which will slow down operations (with collision avoidance strategies having linear complexity, it will lead to a full scan in the worst case and costly rehashing.) On the other hand, if the size of the hash table is chosen too large memory is wasted. Second, since memory is limited the hashing space is restricted to a certain range of values with collisions occurring naturally. Third, hash tables do not support range queries as the hash function arbitrarily maps key values.

Extendible Hash Maps provide a way to combine the advantage of a quick $\mathcal{O}(1)$ lookup with optimized data structure as used in many database systems. Traditionally, B-trees are suggested as an internal data structure for EHMs. We want to take EHMs a step further by outlining a way to decide which internal bucket structure and size should be chosen for a given workload. Using statistics, this strategy can then be easily extended to an online version.

3. HYBRID EXTENDIBLE HASH MAPS

Assume we are given a fixed workload of read and write operations, and knowledge about the read/write ratio of these operations. Then, in our approach we first partition the image of the hash function for a given directory depth (chosen such that the directory ideally fits into one page, i.e. global depth 10-14bit) according to the prefix associated with the global depth of the directory.



a. Extendible Hash Map using static bucket size and simple array for each bucket

b. a. Hybrid Extendible Hash Map using optimized bucket size and a combination of simple arrays, sorted arrays and B^+-trees for buckets

Figure 1: Comparison of traditional extendible Hash Maps with our approach

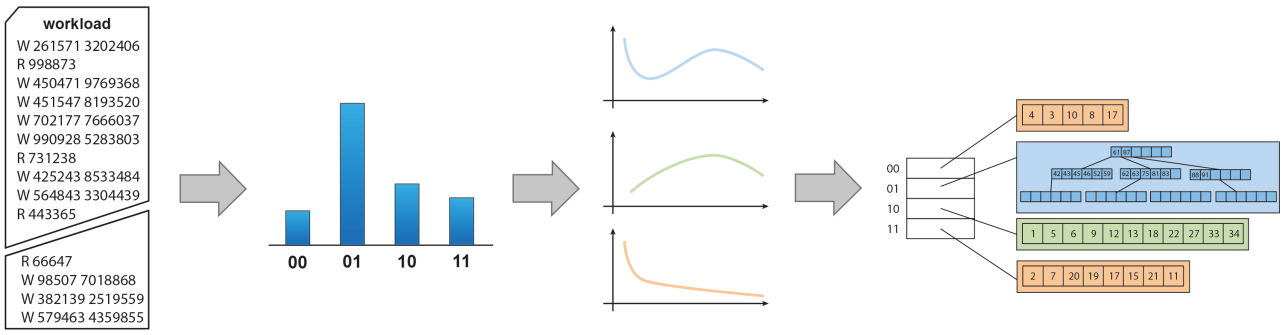


Figure 2: Proposed approach for hybrid extendible hash map construction

For each prefix the ideal data structure and bucket size is then inferred from the distribution. In future work we seek to develop this into an applicable heuristic and will show that such a data structure is able to outperform a static version (cf. Figure 2). In Figure 1 the general structure of a HEHM compared to a traditional EHM is shown. Note that in our preliminary version the size of a bucket associated with a certain prefix is not chosen at runtime but inferred from the fixed workload we are given.

4. EXPERIMENTS

4.1 Implementation

For our experiments we use a custom, locally memory-aware C++ based implementation which is available at https://github.com/LeonhardFS/AdaptiveHashMap. In the implementation we store integer keys and integer values within the buckets(i.e. one entry is 8 bytes). Since we restrict the maximum directory depth, overflowing buckets might occur and are handled via chaining. Using a larger directory depth decreases the risk of overflow bucket occurrence.

For the Java implementation, we use an open-source version of a standard EHM. This implementation has no fixed directory size and uses array lists for both the directory and the internal bucket structure. We use this as a sanity-check for our C++ implementation in case there are any questionable results. The source code can be found here: https://en.wikibooks.org/wiki/Data_Structures/Hash_Tables#A_Java_implementation_of_extendible_hashing.

4.2 Performance Evaluation

We evaluated the performance characteristics of the generic Java and C++ implementations as we varied the number of buckets, and the read-write ratio of a synthetic random data set that we generated specifically for these experiments. These implementations would be the standard EHM data structure with a fixed bucket size. Our hypothesis is that for read-heavy workloads, a low bucket size should be preferable; while the low bucket size would be costly due to the larger number of splits when writing to the EHM, the I/O access time is reduced for reads. For some ratio of reads and writes, we should see the benefit of the faster read access times outweigh the slower writes from having to perform more bucket splits. This hypothesis is central to establishing the feasibility of creating a hybrid data structure that can have a better performance profile than a static bucket size EHM.

Once we established the approximate relationship between read-write ratio, bucket size and performance from a static analysis, we then assessed the impact of including different internal data structures for each bucket (i.e. standard array, sorted array for binary searching, and a B+ tree.) Our hypothesis here is that workloads that favor larger bucket sizes could also benefit from having some more complex internal structure to improve read and write performance. Smaller

bucket sizes likely favor simple internal structures due to the setup costs of complex structures, the smaller differential in run-time between complex and simple structures, and the fewer I/O accesses for simple structures.

4.3 Data set

Each data set had 1 million random actions which could either be a read, or a write, randomly determined according to the read-write ratio. A read-write ratio of 90 percent would mean that 90 percent of the actions would be reads and 10 percent would be writes. In order to focus on reads and writes, we do not include deletions in our workload. For simplicity, we use integers for our data so both the keys and values are integers.

4.4 Methodology and setup

The experiments were generally performed remotely on a dedicated machine provided by the Harvard DAS lab. Some experiments were performed on a local machine and are noted accordingly.

The DAS lab machine was running 64-bit Linux with Intel(R) Xeon(R) CPU E7-4820, 1TB of RAM, 700GB of local storage and a page size of 4KB. The local machine was a dual-core i7 Macbook Pro running MacOSX Yosemite 10.10.5 with 16GB of RAM and 512GB of SSD storage, with a page size of 4KB.

The Java version was run through the command line using Java 1.7 with no additional arguments. The C++ version was compiled with GCC 4.9.2 on the DAS lab computer, and Clang 7.0.2 on the local computer.

In the beginning, the EHM is empty, and is then progressively populated further as more write actions are performed. In the end we measure the total execution time during the main part of the program (i.e. after initialization and setup.) To the extent possible, we load all data into memory in order to minimize I/O overhead.

4.5 Bucket size and read-write ratios

This experiment was a static analysis of the trade-off between bucket size and performance as the proportion of reads and writes in the data set are varied. We tested both the Java and C++ implementations at a read-write ratio of 10 percent, 50 percent, 90 percent, and 99 percent. The initial EHM is empty and is gradually populated. Due to some portability issues and time constraints, the C++ experiments were performed locally. Before testing, the local machine was first restarted and background application usage was minimized.

From figure 3, our results show that the relationship between bucket size and performance is relatively straightforward for data sets which are not heavily skewed towards reads. With low bucket sizes, we see a very sharp increase in the execution time. This is likely due to the large number of splits that need to be made when inserting new elements since the buckets are small. As the number of buckets increase (at least, up to 200 element buckets that we tested,) it appears execution time is steadily improving. The key here is the L-shaped curve of that the graph exhibits for both the C++ and the Java implementation. The exact execution time and the comparison between Java and C++ implementations is largely meaningless here given they were run on different machines, the slower start-up time of the JVM, and the implementation-specific details that could change

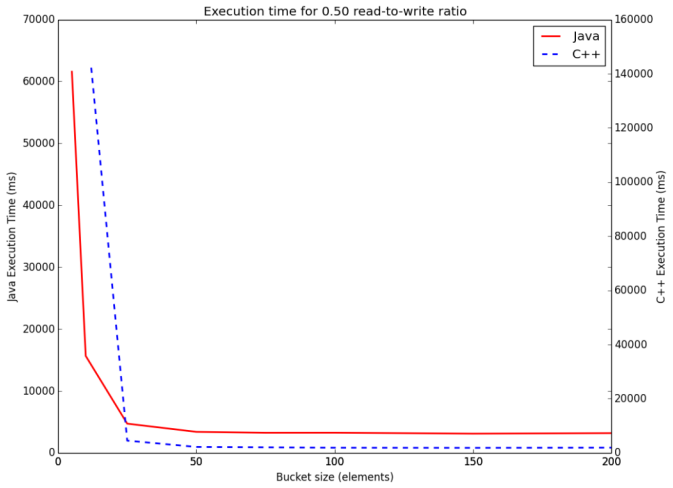


Figure 3: Execution time vs bucket size for 50 percent read-write ratio

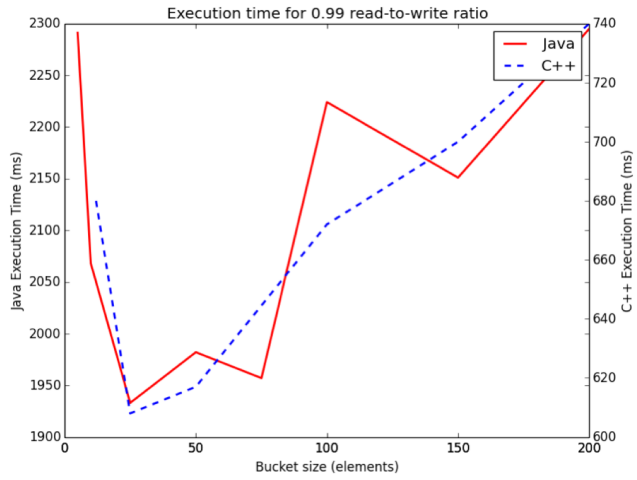


Figure 4: Average execution time per action vs bucket size for 99 percent read-write ratio (C++)

the results. We find this L-shaped curve persists consistently for 10 percent, 50 percent, and 90 percent read-write ratio workloads. It's likely that for even larger bucket sizes, the running time will start to increase as memory reads and bucket splits become more expensive.

The results become more interesting when the read-write ratio reaches 99 percent. As figure 4 reveals, the relationship is no longer simply that a larger bucket size results in faster performance, but rather there is a turning point around a bucket size of 25. This suggests that there can be situations where we can gain performance by specifying a smaller bucket size, rather than a larger one. Note, however, that there is a very steep drop-off in performance when bucket size gets too small, resulting in a very large increase in execution time. This relationship persisted across both the Java and C++ implementations with great similarity; both implementations had the fastest performance for this data set at approximately 25 buckets before performance decreased as bucket size increased.

Using this result, we have validated our hypothesis to some extent; the relationship between bucket size and read-write ratio is not simply that a larger bucket size results in better performance, but rather there can be situations where a smaller bucket size is better. However, it seems there are two caveats to take note of:

1. the drop-off in performance for lower bucket sizes is significant; and
2. the benefits from a lower bucket size are only apparent from workloads with a very high (i.e. 99 percent) read to write ratio.

4.6 Read and write costs in detail

The results from the previous experiment were somewhat surprising. We expected an improvement from having a smaller bucket size for some read heavy workloads, but it took around 99 percent reads to see some evidence of this improvement.

So we looked at identifying the average read and write cost per operation to analyze this trade-off in greater detail. It was clear that writes were creating a significant performance cost when the bucket size was low. In fact, we can see that for very low bucket sizes, the performance suffers greatly. Therefore, understanding the exact cost and benefit of having small bucket sizes is imperative to any hybrid design since we need to know the trade-off and the margin for error. The risk with the hybrid design is that, for example, a smaller bucket size is selected and it turns out the read-write ratio is smaller than expected. Having some idea of how costly getting it wrong can be will inform the correct decision in an adaptive, hybrid design. With this

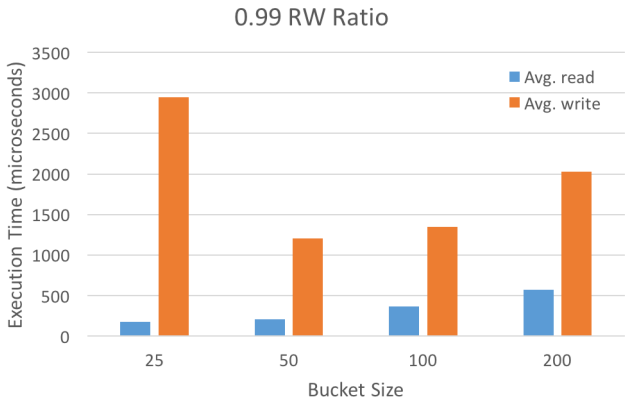


Figure 5: Execution time vs bucket size for 99 percent read-write ratio

goal in mind, we can look at the average execution time for each read and write operation for the 99 percent read-write ratio workload in figure 5. To keep things simple, we just look at the C++ implementation. From figure 4, we saw that a bucket size of 25 was the optimal when it came to performance but noted the sharp decrease in performance when bucket size was smaller than 25. In figure 5, we can see why a 25 bucket size was best; average read time for the lower bucket sizes are notably faster, so the high ratio of reads to writes in the workload offsets the additional cost from expensive write operations. Interestingly, it seems a bucket size of 50 actually has the best average write performance of the four bucket sizes. We argue that with this additional information, in practice, for a similar workload, it may be better to choose a bucket size of 50 for the sake of robustness because of its superior average write performance. It would seem that if the read-write ratio was to decrease even a relatively small amount, performance with a bucket size of 25 could be affected greatly since it appears that average write performance is quite very slow. We propose that there should be some trade-off measure for future adaptive, hybrid designs that assesses the ratio of average read to average write time, versus the potential performance gains.

For example, in figure 5, average write time is 17x average read time for a bucket size of 25, while only 6x for a bucket size of 50. When we compare this to the approximately 10ms improvement over the entire workload, we should probably conclude that a bucket size of 50 may actually better in practice. This is because in practice we value robustness, and may have uncertainties about the workload. Of course, this assessment might change if the workload was much larger such that the absolute performance benefits are greater, or if we have more certainty about the workload.

So the takeaway here is that we cannot just assess absolute performance unless we are dealing with identical workloads. Often, the relative frailty of an EHM configuration may not be apparent. Instead, we should also take into account the relative costs of reads and writes in order to improve the robustness of choices in an adaptive, hybrid design, lest we end up with a very poor performance if our assumptions turn out to be incorrect.

4.7 Internal data structure

The other key optimization we investigated is the internal data structure of each bucket; in particular, we look at a simple array, a sorted array, and a B+ tree. Our hypothesis is that for a given workload, there is an optimum combination of internal bucket data structure and bucket size which should yield the best performance. We also expected that a simple array would probably work best with smaller bucket sizes. Then, as the bucket size increased, the sorted array would probably perform better followed by the B+ tree. The B+ tree in particular would likely require a large bucket size to perform well given the overhead for the data structure and because it tends to work well with larger amounts of data.

For this experiment, we focused on our C++ implementation and ran the experiments on the DAS lab machines. Rather than bucket sizes being defined by the number of integer elements, here we define it in bytes as the B+ tree has overhead so it does not necessarily correspond easily to the number of elements. For the standard linear array and sorted array, each integer is 4 bytes so a bucket size of k bytes can stores $\frac{k}{4}$ elements. The B+ tree uses 1000 entries within a node such that it fits on less than 4 complete pages. Lookups within a node are done using linear search (this can be optimized with binary search.)

Our results in figure 6 show that for a 99 percent read-write ratio workload, we can expect a standard array to perform better for bucket sizes less than around 500, after which the sorted array performs better. The tree requires a large bucket size (minimum of 12,084 bytes) in order to function, but even for large bucket sizes, we note that it still performs around 3 times slower than the sorted array.

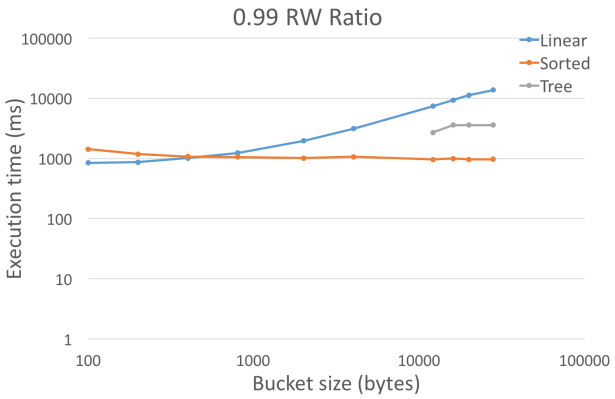


Figure 6: Execution time vs bucket size for different internal data structures at 99 percent read-write ratio

This trend generally repeats itself across the workloads with 50 percent and 10 percent read-write ratios (figures 7 and 8,) but the key difference appears to be the cross-over point between the linear array and the sorted array. In terms of absolute performance, the sorted array was best for the 10 percent and 50 percent workloads, and the standard array with linear lookup was better for the 99 percent workload.

Notably, the optimum bucket size for the sorted array is much larger than for the linear array. This is in-line with what we expected since the benefit of the sorted array is the faster lookup, but this performance boost is only going to be noticeable for large bucket sizes; for smaller bucket sizes, the cost of maintaining the sorted array is likely to outweigh the benefits from a faster lookup since the difference between a $O(N)$ and $O(lg(N))$ lookup is small. This observation also extends to the bucket size at which the sorted array becomes faster than the standard linear array; when the number of writes compared to reads is higher, the overall benefit of a faster read time is diminished while the costs of a slower write time are exacerbated, requiring a larger bucket size to enhance the difference between a logarithmic and linear lookup.

When it comes to the B+tree internal data structure, it finished slower than the sorted array and standard array in every workload. We would expect the B+ tree to be faster for large workloads that require a significant number of reads and writes since:

1. the writes can be performed in logarithmic time compared to the linear time of the sorted array;
2. reads are performed in logarithmic time; and
3. the overhead of the data structure can be amortized over a larger number of operations.

We suspect that the workload simply was not large enough to really see the benefit of using a B+ tree, and would likely perform best if the workload was sufficiently large. However, it does appear that the B+ tree is more robust in the sense that it does not exhibit extreme fluctuations in performance for different bucket sizes, as the linear and sorted arrays do.

Finally, a key takeaway here is that there is a large variance in performance across both the read-write ratio, bucket size, and internal data structure. This bodes well for the possibility of creating an effective hybrid EHM design that will perform significantly better than a standard EHM.

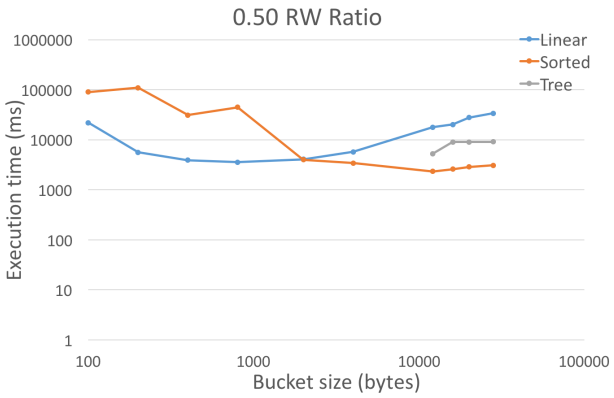


Figure 7: Execution time vs bucket size for different internal data structures at 50 percent read-write ratio

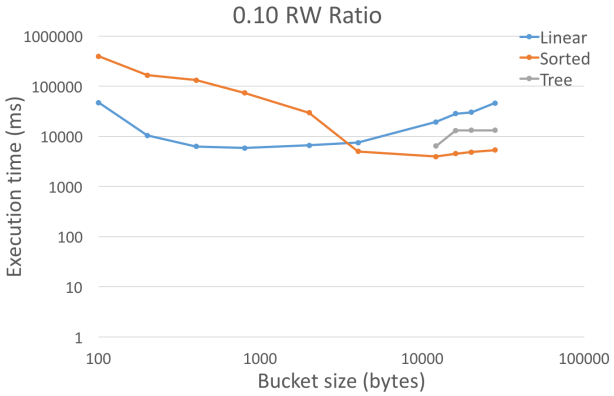


Figure 8: Execution time vs bucket size for different internal data structures at 10 percent read-write ratio

5. DISCUSSION

In this work, we presented a novel approach how to construct a hybrid extendible hash map. A general overview of the proposal is visualized in Figure 2. However, we are still at the beginning of investigating this data structure. So far, we relied on a fixed workflow to construct the hybrid table. To make the approach feasible for application a heuristic is needed that is able to decide on-the-fly which bucket size and which internal data structure to use. Another interesting idea arises when it comes to the heuristic. With the greater goal of designing self-adapting databases or file systems a heuristic could be designed that effectively trades off speed versus memory for the individual buckets. Hence, our work can thus be seen as one building block within the RUM conjecture [**?**].

Especially B-trees in our data structure have expensive split operations (since two trees need to be constructed from the bottom up in our implementation when a split occurs). Thus, maybe using an adapted version based either on prefix

B-trees (while using the suffix of the hash function as the prefix) or developing a split-optimized tree structure could be a rewarding area for further research.

Furthermore, our current implementation of the B-tree allows a minimum filling of 1/2 the node size and linear search within the node. One possible factor that could leverage two different B-tree structures would be to use binary search within a node. We would expect that B-trees with smaller nodes and linear search might be able to outperform the binary search in some use cases allowing for further flexibility within our hybrid hashing approach.

Also, right now we store only integers. With regard to the increasing demand of storage systems for fast access of different sized pictures, our hybrid data structure could be leveraged to provide a page optimized access along fast query processing. The structure then could be used to form the basis of a fast image processing framework.

A key experiment that we would like to investigate in the future is to compare the performance of a manually optimized, hybrid EHM containing different bucket sizes and internal data structures, to a standard EHM. We would use the static analysis that we have performed in this project to inform the manual optimization. This would be an essential step to showing that a hybrid design is worthwhile. We suspect the results will be positive for the hybrid EHM given the large variance in performance across the different bucket sizes and internal data structures; clearly selecting the right combination can have a big impact on performance.

There are more experiments that we think would be useful to investigate. A larger workload that extends beyond 1 million actions could be interesting since the B+ tree internal data structure did not seem to have reached its full potential using the current workload. Also, at the moment, the EHM starts from an empty state and is then progressively populated so the results are likely different to a case where the EHM starts from a partially populated state and then further actions are performed on it. Starting with a partially-populated EHM would likely be more representative of how the data structure would perform in the long run, since we avoid the initial overhead costs (e.g. from splitting buckets.)