Adaptive Hash Tables

Julio Ballista
Stanford University

Maxwell Darling
Stanford University

Michaela Murray
Stanford University

1 Introduction

Although hashing is a fundamental theoretical concept, iterative improvements decade after decade demonstrate that we're still a long way from a perfectly optimal implementation. Fast and efficient hash tables are a highly valued commodity, and they matter now more than ever. The utility of a data structure that can perform inserts, lookups, and deletes in constant-time captivated many and led them to regard hash tables as indispensable tools in practical applications. As usage of hash tables grew, they began to play a significant role in making sure queries were quick and reliable at scale, whether they served as the primary underlying implementation for a database or played a supplementary role [16]. Now, the ubiquity of hash tables in modern-day querying applications demonstrates their success in the commercial sector. Hash tables have also been the subject of numerous research papers: for example, there have been many significant breakthroughs in this field in the past decades, such as Cuckoo hashing and Murmur hashing. However, as we enter a new age of data mining, processing, and storage, hash tables will need additional features and speed-tricks to keep up with the times, especially when new key-index structures like Adaptive Radix Trees and Learned Indexes are looking like more viable options than hash tables. [7,9]. However, as hash tables are the foundation of efficiency, to improve hash tables is to confer a ripple effect of speedups to millions.

Hashing is often treated like a "black-box" [1], where implementation details such as hash scheme and function or the combination thereof are assumed to be optimal for any given task. Thus, some possible optimization vectors have been generally overlooked. The pervasiveness of this false assumption has been recently apparent, wherein two separate papers, each introducing new key-index structures, both naively benchmarked their structures against rudimentary hashing implementations and touted better performance [7,9]. However, each was met with swift responses that demonstrated that "modernized" or better-tailored hash implementations would win out against the proposed structures [1,12].

Now, contrary to the predominant view of hash tables as all-purpose black-boxes, recent research has shown that there is no single one-size-fits-all implementation. Instead, application specifics such as load factor, read/write workload, etc. should be able to determine the optimal implementation, which can significantly improve performance over a randomly (or even strategically) chosen static implementation.

We propose Adaptive Hash Tables, which perform on-the-fly hash table/function switches based on relevant real-time factors, such as load factor, read-write workload, and data distribution. The core idea is that during every resizing of the hash table, metadata on the above factors are examined and the optimal hash function and scheme is chosen according to a heuristic. In our study, our heuristic is a "decision graph" that specifies optimal configuration (according to benchmarks) of hash scheme and function based on five dimensions: data distribution, load factor, dataset size, read/write ratio and unsuccessful lookup ratio [1].

Our ideas differ from current research, as, to our knowledge, there does not exist a hash table that is able to adapt itself to be in its most efficient form. This would require the table to observe its own meta-data, and make decisions as a result, which has yet to be implemented in research.

In sum, we aim to continue the effort of hash table optimization not by introducing a novel algorithm that will replace those before it, but by positing an approach that uses the strengths of the diverse family of pre-existing algorithms.

To evaluate our approach, we measured and compared the efficiency of our adaptive hash table to static hash tables, stratifying across two data distributions (Dense and Sparse), Using these factors, we observed if changing the structure of the generic hash table makes a significant efficiency difference, or if adaptability is simply a better idea in theory, but not practice.

As they are more efficient, if our improved Adaptive Hash Tables are adopted, there would widespread performance gains to be had by individuals and corporations alike. Furthermore, our research serves as the launching point of more potential interest and investigations into this area. Before (and even now), many believe hash tables are a ‘solved problem’ and thus the idea to dynamically alter the structure of hash table implementations in applications has been overlooked. Our research has the potential to showcase how bringing together multiple structures in one adaptable form can prove to be significant and could allow for more exploration in other data structures that are believed to be ‘solved’ or previously unexplored niches for hash tables. Furthermore, this adaptable hash table could only be further improved through other features, such as machine learning to continue resolving the issue of collisions while hashing.

2 Related Works

Fast databases in the present day are more important than ever before. With an unprecedented reliance on data driven by artificial intelligence and cloud computing, we must continue to optimize queries on massive data sets to meet demand. Increasing query performance by even a small constant factor can have a huge ripple effect across large systems and result in huge savings of time and money. With optimization being our main ambition, too, we briefly survey the recent related work in the field - including advances in both hardware and algorithms - before presenting our own approach.

2.1 Hardware

Improving the efficiency of query look-ups is not a new engineering challenge. Past research suggests that query engines could be improved through hardware. There have been attempts to auto generate code that would be most efficient for specific processors, which was achieved through a project known as Hawk. [2]. The underlying assumption here is that data structures cannot further be improved, so we must look to run code that is instead more efficient for specific architectures. This idea was built upon in other research, that suggests that GPU based query processors that use kernel-based execution for queries could also be sped up, as a kernel based approach does not properly utilize all resources available. [3].

There have also been efforts to transform more software based approaches to query into hardware based algorithms. Research has shown that hash functions that rely purely on software is more prone to collisions, jeopardizing the constant time lookup that makes these data structures so efficient . It was found that a hardware approach was generally more efficient. [4]. This research, however, did not look into the actual implementations of said hash tables, though some research has suggested that cuckoo hashing, when running over particular spaces, may have a higher failure probability. [5]. This research lends itself to the idea that certain hash tables may benefit certain sample sizes, but this research does not mention the possibility of creating an algorithm to choose the best table, as our research will attempt to do.

2.2 Hashing Tables Schemes and Functions

Hash tables are essential to database implementations on the algorithmic side, and have only gained popularity in the recent past. As main-memory computing has gained prevalence, tree-based indexes optimized for on-disk applications have fallen out of favor, allowing new hashing implementation which perform well in main-memory like Cuckoo hash [8] and Murmur hashing to become ubiquitous [1]. Additionally, specified research has been done to improve upon current hashing schemes in different contexts like introducing Smart Cuckoo Hashing for cloud storage computing [13] and Cuckoo++ for networking [11,15]. There has also been some more theoretical work in this area as well, such as improving algorithms for cuckoo hashing in a concurrent setting [14].

2.3 Hashing Alternatives

Hashing is not the only option for databases, as evidenced by exciting new research. Perhaps most notably, “Learned indexing” from Google [9] presents a whole new approach to querying, and even more importantly, has opened the door wide open for future research in AI for systems. Additionally, adaptive Radix trees are a clever revival of the long out of style b-trees, which are competitive in performance with hash tables and have unique benefits due to the internal sorting of elements [7]. Interestingly, both the Radix tree and Learned Index papers claimed to have better performance than hash tables. However, when each was tested against the optimized hash implementations, hashing still reigned supreme [6,12].

2.4 Proposal

Upon reviewing the aforementioned work, we observe an apparent ambiguity in determining which database implementations are best for a given task. Thus, our goal is to resolve such ambiguity and strive to find ways to help make choosing an implementation for a given task more clear-cut. Recent research has contributed a decision-making aid in the form of a flowchart to help practitioners choose the most optimal hash scheme and hash function for a given task [1]. We acknowledge this paper's importance, and aim to solidify its findings. Instead of targeting a small, more technical population of database practitioners, our vision is of a tool (adaptive hash table) that can assist the broader audience of database non-experts, including software generalists, students, and researchers, to make an educated decision about what hashing implementation (or otherwise) to use for best performance. Our implementation generalizes to a broader audience due to the nature of automation of hash table efficiency. By having a table abstract the details of choosing the best table, we enable a large range of engineers to have access to the best data structures. The rest of this paper outlines the technical implementations of such, along with our evaluations.

3 Setup

Similar to 'A Seven Dimensional Analysis of Hashing and Its Implications.', our experiments will be single-threaded. All hash tables will be standard maps - they cover both keys and values, and all experiments will exist in main memory. We will be running our experiments on a Google Cloud Linux Instance, with C++ being our language of choice for implementing the tables. Our overarching goal is to replicate the testing environment in 'A Seven Dimensional Analysis of Hashing and Its Implications' as closely as possible to ensure that our results are comparable and are free from unexpected sources of error.

3.1 Test Data Distributions

Every indexed key in our table will be 64-bit integer in the range $[1,\ 2^{64}-1]$. The dataset we gradually feed to the hash table will have distributions as described in section 4.3 of [1], and we will classify tests as either using a 'Dense' dataset or a 'Sparse' dataset. We know through their findings that data distribution has an impact on the best hash table to use, so this would be a great metric to test if our adaptive table can figure out a distribution based on a dataset, and adapt to its findings. We define a 'Dense' dataset to be every key from $[1:N]->\{1,2,3,...n\}$. This data is tightly compact, and is substantially more predictable for hash functions to deal with. We define our 'Sparse' dataset to be a random set of keys from $[1:2^{64}-1]$. This will lead to greater variations of the data, which can put our tables to the test. These data distributions models come strictly from 'A Seven Dimensional Analysis of Hashing and Its Implications' and will be used to better simulate the tests run by the aforementioned paper.

3.2 Adaptive Hash Table Implementation

Fundamentally, the adaptive hash table class is simply a wrapper class that at any point in time maintains a single, active instance of a hash table object [1]. Additionally, the class collects metadata based on queries made over time, such as the number of successful and unsuccessful lookups, the relative density of the dataset in the hash table at a particular instance in time, and information about the load factor of keys. Two new functionality our wrapper class introduces to the hash table implementations are two functions: one assess whether it's time to switch, and the other manually switches the table. The former function analyzes the lookup ratio, density, and load factor at a particular point in time and returns a data structure with all the updated metrics. The lookup ratio is will be calculated and updated as one starts looking up information in the table. This is key because if the lookup ratio is largely unsuccessful (below 50%) then the table might not be optimal for the given situation. The density is defined as follows: We assume a fixed number of keys n at the beginning and double the number of assumed keys each time we resize the table. Thus, our implementation will tend towards a sparse dataset since we started with a fixed number of keys and no data, and a sparse dataset lends itself better towards less hash collisions. Such metrics can then be used by the latter function to either switch to a different table or keep the current implementation. If it is optimal to switch implementations, our current setup automates the flowchart [1] and depending on which table is selected, a transferHash function we inserted into the relevant table files will transfer all values from one table to another. Importantly, the difference our table makes is that once enough inserts are made and the table determines it must dynamically resize (by a factor of two), we forgo the standard procedure of rehashing immediately. Instead, the possible payoff of altering the current hash table implementation is assessed. and then the final decision for the optimal implementation is chosen via a heuristic.

3.3 Static Hash Table Implementation

Since we have access to the source code used in 'A Seven Dimensional Analysis of Hashing and Its Implications', we have decided to keep all hash table implementations as consistent as possible with the original code used in the paper. This mitigates the possibility of unintended speed variations, which is vital in order for our results to be comparable. However, since we are building a complicated class structure that must be able to perform adaptations on the fly, edits had to be made to the original implementations to make sure they fit with our vision of the Adaptive Hash Table class. In particular, the pre-existing resizing procedures were entirely removed to integrate with our system, yet the overall resizing method (dynamic resizing by a factor of 2) was preserved. Additionally, insignificant changes to variables and other class-dependencies were removed for smooth integration. The hash table implementations we focused on were Linear Probing and Chained Hashing, and for both, we used the multiply shift hash function since, according to our parent paper, multiply shift was the most optimal hash function across the board for all different hash schemes and data situations. Since each hash table implementation was made in isolation to each other, we had to make some small changes to each code base to make them easily generalizable. Each of these hash table implementations are different and yield unique performance benefits depending on the particular situation, including the type of data distribution and the ratio of successful lookups. Our adaptive table should be able to recognize when one table's inefficiencies start outweighing its benefits, and promptly switch to a different implementation. [1]

3.4 Hash Function: Multiply Shift

Based on the research from the authors of 'A seven Dimensional Analysis of Hashing and Its Implication', we decided that the most relevant and useful hash function in all scenarios was the multiply shift hash function. In the flowchart from the paper, the most optimal choices used multiply shift as their primary hash function and to reduce complexity, we decided to follow suite.

4 Evaluation

With the creation of our adaptive hash table, our primary goal is to compare its performance against traditional hash tables. Our hope is to identify non-trivial use cases in which using an adaptive hash table for a given task would be more performant than a traditional hash table. To do so, the performance of adaptive hash tables and traditional hash tables will be measured and compared across a number of experiments, in which properties of the data (eg. density, size) and use-case specifics (eg. ratio of reads/write, ratio of successful queries) will be varied. The two primary measurements of interest are the number of insertions per unit time, and the number of lookups per unit time. Further, in the experiments that follow, we consider only a 'RW' (read-write) workload and postpone the analysis of the 'WORM' (Write Once, Read Many) workload for future work. The fundamental reason is that our adaptive hash table requires dynamic resizing to occur, which obviously requires insertions to be made continually over the lifespan of the table, as opposed to all at once. We do not, however, discount the value of optimizing for the 'WORM' workload, and have therefore given it consideration in the discussion section.

4.1 Measurement and Analysis

Two distinct types of experiments were conducted to measure the hash tables' performance: 1. the main experiment, in which a set of both insertions and lookups were performed over many intervals with a resize ocurring at the end of each, and 2. supplementary experiments that directly measured the speed of individual operations (insert, lookup, resize) in which tables did not resize (for insert and lookup), or resized only once (for resize). Greater detail is provided in section 4.2 for the main experiment and section 4.3 for supplementary experiments.

Additionally, four distinct types of hash tables were analyzed. As for traditional hash tables, Chained and Linear Probing were used. Furthermore, two implementations of adaptive hash tables were used. In all experiments, the standard 'wrapper-class' implementation that is described above is used. However, in the main experiment, an additional 'lightweight' implementation was also considered, whose purpose was to reveal the performance upper bound by "cheating", and is therefore not usable in practice. In specific, instead of a wrapper class that contains an abstract class pointer to the current hash table and performs it's operations by passing them on to that abstract object (1 layer of indirection), the 'lightweight' implementation is simply the abstract class object itself with adaptive resizing and other opertations performed manually, thus having no indirection and mimicking the behavior of it's subclasses due to runtime polymorhpism. Further detail on the time savings is provided in section 4.3 (insertion time overheads). To reiterate, the lightweight table serves as an upper bound that the standard adaptive table implementation should strive to achieve by reducing overhead with further optimization.

Lastly, it should be noted that the main experiment accounted for a single dimension: un/successful lookup ratio. Since we knew a priori that the most optimal hash tables under low and high unsuccessful lookup ratio are linear probing and chained hashing respectively [1], we took advantage of this in our experiment by initializing our adaptive hash table with the linear probing implementation and initially performing only successful lookups, only to gradually increase the number of failed lookups until the adaptive table determined it should adapt to chained to maximize performance. A procedure for adapting according to data density - the other most salient dimension - was being developed concurrently, but was not completed in time (see section 5 for plans for future work).

4.2 Results: Main Experiment

In this experiment, the performance of four hash tables was measured over four intervals of operations consisting of both lookups and insertions. Linear Probing (LP), Chained (CH), the adaptive table (AHT2), and the 'lightweight' adaptive table (AHT1) were analyzed (for details on the latter, see section 4.1). The goal of the experiment was to produce conditions in which an adaptive hash table would outperform traditional hash tables. In this case, such a "condition" was to gradually increase the ratio of unsuccessful lookups, and have the adaptive table switch from it's LP initialization to CH. In specific, each table was initialized with a capacity of 2^{18} keys. Then, a mix of insertions and lookups were performed in each interval, in which the % of unsuccessful lookups increased by 25% each time, from 0% to 75% across 4 intervals. Further, the ratio of lookups to inserts varied during internal testing, but for the experiment presented, such ratio is 24 (ie 24 lookps per 1 insert). As an example, in the first interval, $24 * 2^{17}$ lookups (all successful) and 2^{17} keys inserts were performed, leading to a resize due to the maximum load factor of 0.5 being reached. In subsequent intervals, the table capacity is doubled each time, so the number of inserts doubles accordingly to force a resize on the last insert of each interval. Time was measured at the end of each interval both before and after resizing (the vertical lines in figure 3 represent the time taken to resize). In this experiment, the adaptive hash table was programmed to adaptively resize to CH when the unsuccessful lookup ratio met or exceeded 50%, which is in accordance with the previous literature[1]. A total of 5 trials were performed, and their results were averaged (although variance was quite low).

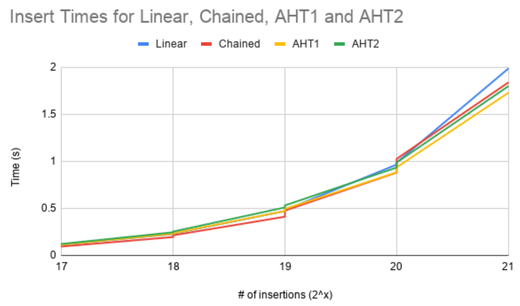

Figure 3. lookups & inserts over four intervals

As seen in figure 3, AHT1 and AHT2 outperform the traditional implementations by a small margin. We expect this to be explained by our a prior knowledge about CH and LP. Comparing the red and blue lines in figure 3, we do indeed see that CH clearly outperforms LP in the rightmost interval when unsuccessful lookups are 75%. However, interestingly, we also see that CH performs slightly better on the left side of the graph when unsuccessful lookups are few. This is curious, as experiments from previous literature indicates that LP should have a slight edge over CH for lookups at low unsuccessful lookup ratio (and a big advantage in general for insertions). This result is contrary to that. One possible explanation for this is that the code we inherited for the CH and LP implementations provided are modified versions of that used in the original set of experiments, as our protocol for carrying out experiments is adapted exactly from the original. In any case, we see that both AHT implenentations outperform CH. The reason? Despite CH being slightly faster on the left side of the table, the key point is when a resize happens at 2^{20} insertions, wherein the CH table takes a very long time, and lags behind both AHT tables, whose adaptive resize from LP to CH is faster than the standard CH resize. From that point, we can see that those three tables perform equally well with the remainder of insertions, shown by the parallelism of the 3 corresponding lines. Ultimately, the victory of the AHT is a close one, and attributed to a somewhat messy intermingling of performance differences across the insert, lookup, and resize operations. Thus, to shed light further on each of these operations, we present a supplementary set of experiments that examined each type of operation in the next section. For an overall discussion, see the discussion section 5.

4.3 Results: Supplementary Experiments

In this section, we discuss the differences between insertion, lookup, and resize operations. We'll first examine insertions.

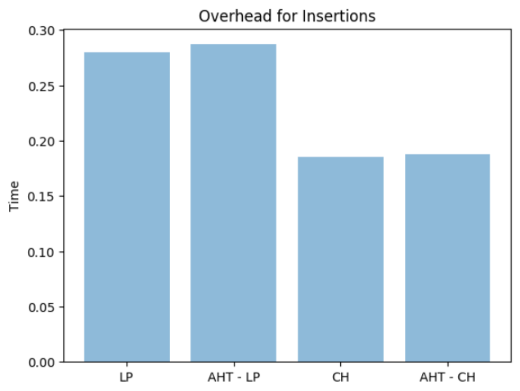

Figure 4. Insertion Time Overheads

For this experiment, insertion time was measured for 2^{21} insertions for each type of table. Figure 4 illuminates

2 key points. First, we can see clearly that CH has faster insertions than LP, which is again contrary to what was proposed in the original paper. Additionally, we can see the effect of overhead on performance that comes with the AHT table. In both cases, the AHT implementation introduces a small additional time cost. This may look small, but when it compounds over millions of operations, can lead to non-trivial performance differences.

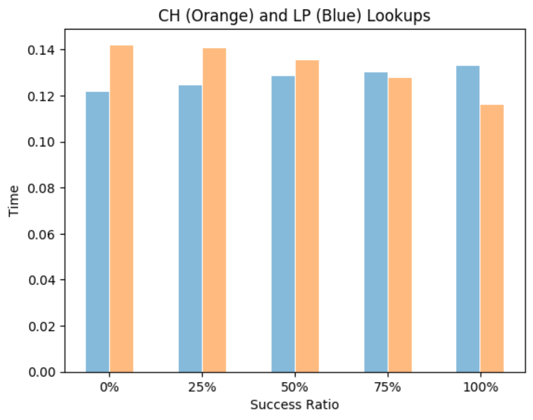

Figure 5. Lookup times

Now, we'll discuss lookups. Figure 5 shows a comparison of the lookup times for the basic CH and LP tables. In these experiments, tables were filled with 2^{18} keys and a set of 2^{18} lookups were performed, with varying unsuccessful lookup rates. From figure 5, we can tell that LP gets worse with the number of unsuccessful lookups, and CH gets better, which is in line with the original study. Furthermore, we see that LP starts off more optimal, and that the point of eclipse occurs at around 50-75%, which is also consistent with the study. Overall, we can be happy with these results as they are consistent with the previous study. Lookup times were not included for the AHT tables, as there was no difference, which is what was expected given their implementation which has little to no excess instructions between AHT1 and AHT2.

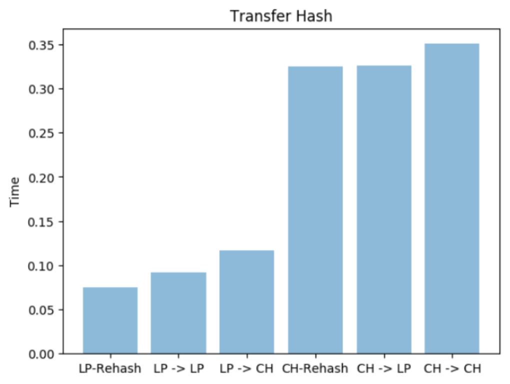

Figure 6. Transfer Hash

Further, in figure 6, we see the results of an experiment measuring rehashing times. In total, 6 types of rehashing were considered: basic CH and LP rehashing, and then our own adaptive resizing methods in four scenarios: LP to LP, LP to CH, CH to LP, and CH to CH. Note that for this experiment, each table resized to a capacity of 2^{22} from 2^{21}. We can glean two observations from figure 6. First, in comparing standard LP and LP-LP, and likewise standard CH to CH-CH, we see that our custom adaptive method does indeed have a small overhead. This is as expected, as it is implemented fairly naively by simply iterating over all the keys in the existing table, and calling the insert method on each for the new table. This is opposed to the standard resize implementations that are slightly more optimal by taking advantage of internal class structure. Secondly, figure 6 makes it clear that LP takes a significantly shorted time to resize. As resizing fundamentally consists of iterating over keys and inserting them, and since CH has quicker insertion times, we can surmise that the speed of iterating over the CH table is far slower than that of the LP table, which is likely attributed to following pointers between chained entries as opposed to directly indexing into an array in open addressing. It is this very resize time difference that makes LP somewhat competitive with CH in the main experiment.

5 Discussion

The following discussion section explains the implications of our results overall. Firstly, although our adaptive hash tables slightly outperformed their counterparts in the main experiment, we cannot endorse their use. Firstly, the reproducibility issue of CH vs LP lookup times in the main experiment indicates that further corroboration is needed between us and the previous authors before any concrete claims are made. Asides from that, there are further key points we'd like to make about our results. First is the consideration of resize times. The previous literature did not examine the differences in resize time for each implementation, but we were surprised to find this to be absolutely critical for overall performance. For example, when the insertion to lookup ratio is 1 instead of 24 (as it was in the main experiment), LP finishes on average in 0.37 seconds, far exceeding CH, AHT1, and AHT2, with times 0.75s, 0.59s, and 0.61s respectively. In this case, despite slower insert and lookup times, the super-short resize time of LP leads it to dominate the other tables, which must incur Chained hashing's slow resize time (3 times for CH, and once for AHT 1 and 2). LP only starts to get outperformed when the lookups to insertion ratio is increased, as it's worse performance on lookups starts starts to bear greater consequences. This discovery was quite a suprise, since at the outset of our research we were certain we only needed to consider the tradeoffs between lookups and insertion times.

5.1 Limitations

Overall, there are two primary limitations we would like to address: the dependency on outside hash table classes and the relatively small scope of our class and test harness. While using outside hash tables does help us standardize results against already published and approved data, it does have some limitations with regards to the performance and reliability of our class. Since we did not go through and implement each class ourselves, there are some discrepancies and non-standardization between implementations that does make it hard to pin down exact results, and in the end, unfairly contribute to the overhead in several key functions such as insert and lookup. In the future, possible changes such as implementing our own classes or creating one big class altogether might alleviate these limitations. To address the second point, while our implementation only runs over a limited number of dimensions and hash table types, it is generally more advisable to run tests over a limited number of parameters and types at the beginning to verify one's own assumptions about hypothesis in question. If a certain hypothesis can be easily disproven early on, then the question at hand might not be worth pursuing in the long run.

6 Conclusion

The end goal of this research is to introduce a more optimized hash table that can have real benefit in practical settings and the real world. In this first attempt, we made significant progress, coming up with an implementation - although limited to a single dimension - that can adapt and lead to performance benefits in narrow cases. However, we were caught off guard by the massive importance of resize times, and thus will address this head-on in future research. Consideraton of resize times on top of the insert and lookup benchmarks laid out in prior research make the problem even more complicated, but still absolutely worthwhile. Above all, we hope that this paper garners recognition for the very idea of an adaptive hash table, an idea which we think is very exciting and has promise. With continued effort and collaboration with other researchers, we look forward seeing to how far we can take the adaptive hash table.

7 References

[1] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. Proc. VLDB Endow. 9, 3 (November 2015), 96-107.
DOI=http://dx.doi.org/10.14778/2850583.2850585

[2] Martin Dietzfelbinger and Ulf Schellbach. 2009. On risks of using cuckoo hashing with simple universal hash classes. In Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms (SODA '09). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 795-804.

[3] Sebastian BreB, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. The VLDB Journal 27, 6 (December 2018), 797-822. DOI: https://doi.org/10.1007/s00778-018-0512-y

[4] K. Kara and G. Alonso, "Fast and robust hashing for database operators," 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, 2016, pp. 1-4. doi: 10.1109/FPL.2016.7577353

[5] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based Pipelined Query Processing Engine. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). ACM, New York, NY, USA, 1935-1950. DOI: https://doi.org/10.1145/2882903.2915224

[6] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In 31st IEEE ICDE, April 2015.

[7] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In 29th IEEE ICDE, pages 38–49, April 2013.

[8] R. Pagh and F. F. Rodler. Cuckoo hashing. Journal of Algorithms, 51(2):122–144, 2004

[9] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18). ACM, New York, NY, USA, 489-504. DOI: https://doi.org/10.1145/3183713.3196909

[10] Mark Slee, Aditya Agarwal,and Marc Kwiatkowsk. Thrift: Scalable Cross-Language Services Implementation. 2007. Facebook.

[11] Nicolas Le Scouarnec. 2018. Cuckoo++ hash tables: high-performance hash tables for networking applications. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems (ANCS '18). ACM, New York, NY, USA, 41-54. DOI: https://doi.org/10.1145/3230718.3232629

[12] Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, Matei Zaharia. Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes. Stanford University. 2018.

[13] Yuanyuan Sun, Yu Hua, Qiuyu Li, Shunde Cao, and Pengfei Zuo. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. Usenix Association. 2017 USENIX Annual Technical Conference (USENIX ATC '17). 2017.

[14] Xiaozhou Li, David G. Andersen, Michael

Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14). ACM, New York, NY, USA, Article 27, 14 pages. DOI: https://doi.org/10.1145/2592798.2592820

[15] Fumito Yamaguchi and Hiroaki Nishi. Hardware-based hash functions for network applications. IEEE. 2013 19th IEEE International Conference on Networks (ICON). 2013.

[16] MySQL 8.0 Reference Manual: 15.5.3 Adaptive Hash Index. MySQL, dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html.