When Are Learned Models Better Than Hash Functions? (Extended Abstracts)

Ibrahim Sabek* MIT CSAIL sabek@mit.edu
Kapil Vaidya* MIT CSAIL kapilv@mit.edu
Dominik Horn TUM dominik.horn@tum.de
Andreas Kipf MIT CSAIL kipf@mit.edu
Tim Kraska MIT CSAIL kraska@mit.edu

ABSTRACT

In this work, we aim to study when learned models are better hash functions, particular for hash-maps. We use lightweight piece-wise linear models to replace the hash functions as they have small inference times and are sufficiently general to capture complex distributions. We analyze the learned models in terms of: the model inference time and the number of collisions. Surprisingly, we found that learned models are not much slower to compute than hash functions if optimized correctly. However, it turns out that learned models can only reduce the number of collisions (i.e., the number of times different keys have the same hash value) if the model is able to over-fit to the data; otherwise, it can not be better than a typical hash function. Hence, how much better a learned model is in avoiding collisions highly depends on the data and the ability of the model to over-fit. To evaluate the effectiveness of learned models, we used them as hash functions in the bucket chaining and Cuckoo hash tables. For bucket chaining hash table, we found that learned models can achieve 30% smaller sizes and 10% lower probe latency. For Cuckoo hash tables, in some datasets, learned models can achieve a small lookup time benefit. In summary, we found that learned models can indeed outperform hash functions but only for certain data distributions and with a limited margin.

AIDB Workshop Reference Format:
Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When Are Learned Models Better Than Hash Functions?. *AIDB* 2021.

1. INTRODUCTION

Hashing is a fundamental operation in computer science and commonly used in databases [15]. They are mainly used to accelerate point queries, perform joins and grouping, etc. (e.g., [3, 7]). In hash tables, a key is mapped to a location in constant time (i.e., $O(1)$). Compared to the traditional tree-structured main-memory indexes, hash tables have been proven to be much faster for point queries. Meanwhile, a lot of data structures and algorithms are recently being enhanced by learned models (e.g., [11, 12]). These learned structures can outperform their traditional counterparts on practical workloads. One idea the authors of [11] introduced is using learned models instead of hash functions, and was supported that by some empirical evidence. In this paper, we aim to study, in more details, when learned models are better than hash functions, particular for applications like hash-maps. We primarily consider piece-wise linear models in our analysis as they have small inference times and are sufficiently general to capture complex distributions.

Our study investigates the performance of learned models in terms of: *the number of collisions*, and *the computation time*. A collision between two keys occurs when they have the same hash value. Some hash functions are extremely fast to compute, yet, they might suffer from a considerable collision rate in some scenarios (e.g., Multiply-shift [5]). In general, there are known theoretical lower bounds on the number of collisions achieved by hash functions. We observed that learned models might be able to do better than these lower bounds and outperform hash functions. In particular, it turns out that the amount of collisions for learned models is dependent on the data distribution.

Regarding computation time, we empirically found that learned models are slower to compute than most hash functions due to the cache miss overhead from randomly accessing the model's parameters. However, with the help of vectorization and prefetching-optimized inter-task parallelism (e.g., AMAC [10]), the learned models computation time can come quite close to its hashing counterpart (around 2 ns difference using models with moderate size).

To show the effect of using learned models within hashing applications, we built bucket chaining and Cuckoo hash tables using two efficient models, namely RMI [11] and RadixSpline [9], instead of hash functions. Typically, in Cuckoo hashing, a single hash function is used to extract two hash sequences. In our experiments, we computed one hash sequence using the learned model and the other using the hash function. We empirically evaluate the performance of these altered hash tables with various real-world and synthetic datasets. For bucket chaining, we found that learned models can achieve 30% smaller hash tables and 10% lower probe latency. For Cuckoo hashing, in some datasets, learned models can increase the ratio of keys stored in their primary locations (primary key ratio) by around 10% and a small lookup time benefit in the probe phase.

2. BACKGROUND

Hash Functions and Tables. Murmur [15] and XXH3 [4] are among the most widely-used hash functions, which have

*Both authors have equal contributions and their names are sorted alphabetically.

3rd International Workshop on Applied AI for Database Systems and Applications (AIDB'21), August 20, 2021, Copenhagen, Denmark.

good balance between computation time and collision rates. They are implemented with arithmetic (e.g., multiply, add) and logical (e.g., shift, XOR) operations. However, XXH3 is specifically designed for streaming data. AquaHash [16] is another popular hash function that leverages Advanced Encryption Standard (AES) instructions [1]. In general, hashing schemes for handling collisions are categorized into two main categories: chaining and open addressing. Bucket chaining [3] is a standard hash table implementation that follows the chaining scheme. It contains a set of n buckets, where each bucket has a pre-allocated array of s entries. On an insert, once a collision occurs, the item is inserted in the current available entry in its corresponding bucket. If the current bucket is already filled up, a new one is created, pre-allocated and chained to it. For open addressing scheme, Cuckoo hash table [14] has become the recent state-of-the-art. Every item has two possible locations: its primary and its secondary bucket. When inserting an item and its primary bucket is full, it gets placed into its secondary bucket. If the secondary bucket is also full, a random item is kicked from the bucket and is placed into its alternative location (balanced kicking). In contrast, biased kicking [8] prefers kicking items that reside in their secondary buckets. The idea behind this is to increase the ratio of items in their primary buckets (primary ratio) and hence improve performance for positive lookups. Typically, Cuckoo hashing is implemented with two independent hash functions.

Learned Models. Recently, the idea of using learned models to predict the location of keys in datasets has gained a great attention in the database community [11]. RMI [11] was the first proposed index that uses multi-stage learned models. In RMI, the root model gives an initial prediction of the CDF for a specific key. Then, this prediction is recursively refined by more accurate models in the subsequent stages. Interestingly, the authors of [11] also discussed the idea of using CDF-based learned models as order-preserving hash functions, which is the main scope of this paper. An interesting index that followed RMI, namely RadixSpline [9], employs a radix table to quickly find the two spline points that approximate the CDF for a specific key. Then, linear interpolation between the retrieved spline points is used to locate the key. In this paper, we only focus on piece-wise linear models that are built using a set of line segments, where each segment is represented by a slope and an intercept. Both RMI and RadixSpline can be considered as piece-wise linear models, which are just trained/created differently.

3. ANALYSIS OF LEARNED MODELS

3.1 Can Learned Models Cause Less Collisions than Traditional Hash Functions?

In this section, we first characterize collisions and then use this to identify/analyze factors affecting collisions for learned models and hash functions. This analysis helps us to characterize situations where learned models outperform hash functions.

Notation. We consider the task of mapping N keys to N locations for ease of analysis. $x_0, x_1, ...$ is the sorted array of N keys ($x_i <= x_{i+1}$) and $y_0, y_1, ..$, where $y_i \in [0, N-1]$, is the corresponding sorted array of output values ($y_j = f(x_i)$) where f is a learned model or hash function) such that $y_i <= y_{i+1}$. Note that x_i does not necessarily relate to y_i, x_i's and y_i's are just sorted versions of the original input keys and output locations. For learned models, y_i's are continuous values and the precise output location is the closest integer to y_i's. The sorted output values generate a set of gaps $g_1, g_2, ...$ such that $y_i = \left(\sum_{t=1}^{i} g_t\right) + y_0$. These gaps form a distribution G with a probability density function (PDF) f_G. The discussion and analysis in the rest of this section support the following points:

- Collisions are dependent on the gaps between consecutive sorted output values (g_i's).
- For piece-wise linear models, the number of collisions is dependent on the key distribution, specifically the gaps between consecutive sorted keys ($x_i - x_{i-1}$). Higher variation in the distribution of gaps between keys leads to more collisions. Having more linear models improves the accuracy but may not reduce the collisions.
- Collisions for a good hash function are independent of the input key distribution (The distribution of x_i's).

Characterizing Collisions. If two keys are mapped to the same location, then there is a collision. The key insight regarding collisions is that the collisions depend on the gaps between consecutive output values ($y_i - y_{i-1}$). If the gap between two consecutive output values is greater than one ($y_i - y_{i-1} \geq 1$), then they would definitely be placed in separate locations. On the other hand, if the the gap is smaller than one ($y_i - y_{i-1} \leq 1$), they may be mapped to the same location depending on the location boundary. In addition, the smaller the gap value the more the probability of the keys falling in the same location.

The gap values are constrained by the condition that the sum of all the gaps should be less than $(N-1)$[1], and thus, the mean gap value turns out to be less than or equal to one ($E[G] \leq 1$). Ideally, we would want all the gaps to be exactly equal to one as this leads to zero collisions and also satisfies the constraint. Qualitatively speaking, any increase in the variance of gap distribution G leads to an increase in the number of gaps below value one and thus, a subsequent increase in the number of collisions.

Collisions for Linear Models. The output distribution for linear models is dependent on the input distribution. Linear operations scale and offset the input values to obtain the output. For sorted input values $(x_0, x_1, x_2, ..)$, a simple linear model ($y = m * x + b$) will just scale the gaps of the input ($g_i = y_{i+1} - y_i = (x_{i+1} - x_i) * m$). For piece-wise linear models, the gap distribution of the output values G is a scaled version of the input. The scaling will be such that the mean of G is less than equal to one ($E[G] \leq 1$). If the input gap distribution has higher variance, this would be propagated to G, leading to more collisions.

Next, we qualitatively argue why more models do not necessarily reduce the number of collisions. Suppose the input data was generated using a gap distribution H with corresponding PDF f_H. Piece-wise linear models would simply scale different ranges of the input and thus, the corresponding output gap distribution would just be a scaled version of f_H. Increasing the number of models does not alter the gap distribution of the output values and thus, the number of collisions stays the same. In an extreme case, when the number of models is close to the number of keys, then the collisions would be low but the space overhead would make the structure practically unusable.

[1]Sum of gaps is: $\sum_{t=1}^{N-1}(y_t - y_{t-1}) = y_{N-1} - y_0 \leq N-1$

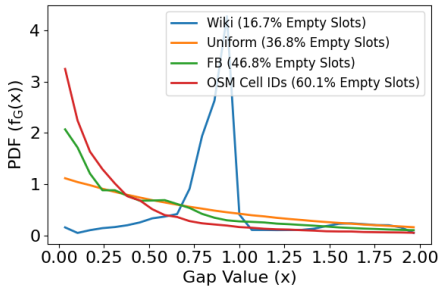


Figure 1: Gap distribution and its effect on collisions

Here, we visualize the gap distribution of the output values for various datasets and the corresponding proportion of empty slots. We used piece-wise linear models for various datasets and obtained the output $y_i's$ (CDF) values for them. In Figure 1, we show the PDF of the gap distribution and the proportion of empty slots for three real datasets from [13] and a synthetic uniform one. Clearly, the gap distribution is much more predictable for *wiki* than for uniform, *fb* and *osm*. *wiki* has a gap distribution concentrated more towards one and so ends up having the least number of empty slots. *osm* tends to have a lot of gaps concentrated towards zero and ends up with the most empty slots. We provide a formal analysis for the collisions and its relation with gaps distribution in Appendix A.

Collisions for Hash Functions. In case of a good hash function (e.g., Murmur), the output values will be uniformly distributed in the range $[0, N-1]$ irrespective of the input distribution. Therefore, the gap distribution of the output values is a fixed distribution which corresponds to the uniform case in Figure 1.

Summary. If the input keys are generated from a distribution, then the CDF of the distribution maps the data uniformly randomly in the range $[0, 1]$. Hence, the CDF will behave as an order-preserving hash function in a hash table. A learned model that approximates this underlying distribution would only be as effective as a hash function in terms of collisions. If the data is generated in an incremental time series fashion $(x_0, x_1 = x_0 + g_0, x_2 = x_1 + g_1, ...)$, the predictability of the gaps determines the amount of collisions. In certain cases, like the *wiki* distribution, a learned model can lead to fewer collisions. Auto generated IDs with some deletions are the other common case where learned models can beat hash functions.

3.2 Can Learned Models be Computationally as Fast as Traditional Hash Functions?

In this section, we first present the computation bottleneck of using learned models as hash functions. Then, we discuss the opportunity of alleviating such bottleneck by using vectorization (i.e., Single Instruction Multiple Data (SIMD) instructions) and prefetching-optimized inter-task parallelism techniques (e.g., AMAC [10]).

Cache Misses Overhead. The computation of traditional hash functions is fast. It usually includes the execution of arithmetic, logical, and shifting operations (e.g., Multiply-shift [5], and Murmur [15]). In contrast, using learned models, like RMI, as a hash function incurs higher latency. This is because, although the inference computation of these models (which is basically the hashing computation in our case) is completely based on arithmetic operations (e.g., add, multiply, max), there is an additional overhead in accessing the model parameters (e.g., intercepts and slopes) that will be used during the computation. This overhead significantly increases if the model size becomes large as its parameters will not completely fit in the cache, and randomly accessing them from the memory will incur many cache misses.

Performance Gain via SIMD. Interestingly, vectorizing the computation in learned models is *more efficient* than vectorizing some hash functions *as long as the model parameters are kept in the cache*. To backup this claim, we micro-benchmarked the throughput of hashing 128 million 64-bit integer keys using a single-threaded AVX512 SIMD implementation for both Murmur [2] and 2-levels RMI model [11], running on a machine with Intel(R) Xeon(R) processors (Skylake architecture). We made sure that all models' parameters are fully cached by building an RMI with a total of 5 linear models only (1 root model, and 4 models in the second level). Our results showed that the hashing throughputs for vectorized RMI and vectorized Murmur are 1000 and 800 million keys/sec, respectively. This is expected because, *with ignoring the effect of parameters' cache misses*, the inference computation (i.e., hashing) in RMI heavily relies on fast comparison (e.g., min/max) and fused instructions[2] (e.g., fmadd), each has a throughput of 2 instructions/cycle [6]. On the other hand, 60% of the total instructions needed in the Murmur computation have a throughput of 1 instruction/cycle or less, such as logical shift (1 instruction/cycle) and multiplication (0.66 instruction/cycle).

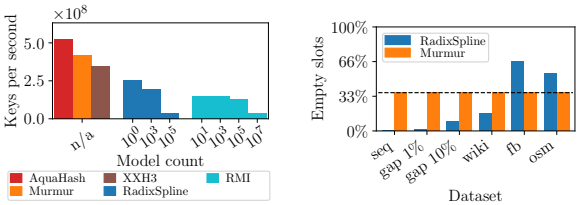
Performance Gain via AMAC. As previously mentioned, the superiority of vectorized learned models quickly diminishes when we have large models, which is a typical case in real settings. In this case, the model parameters are frequently accessed from memory and not from cache. Even if some of the requested parameters from a vectorized instruction hit in cache, the instruction cannot proceed until cache misses of the other parameters in the vector are resolved. Obviously, direct software prefetching is not a feasible solution to this issue, and will completely stall the performance, because the model parameters require immediate memory access. Therefore, we propose to hide the cache misses latency by combining the vectorized learned models with a widely-used prefetching-optimized inter-task parallelism technique, namely AMAC [10]. This helps in making the overall latency of vectorized learned models very close to traditional hash functions as shown in our evaluation (Section 4). Appendix B shows our proposed batch-oriented hash function that combines the benefits of SIMD and AMAC with learned models.

4. EVALUATION

For the experiments, we use three real datasets from [13], namely *wiki*, *osm*, and *fb*, in addition to three variations of a synthetic sequential dataset with different x% elements removed randomly (x={0, 1, 10}). Each real or synthetic dataset has around 200 million 64-bit keys. We de-duplicate the real datasets before using them. We use AquaHash [16], XXH3 [4] and Murmur [2] with fast modulo reduction[3] as

[2]Intel(R) Xeon(R) Gold 6230 processor has two physical AVX512 FMA units.

[3]Modulo reduction is based on efficient integer division (https://libdivide.com).



(a) Median keys per second (b) Empty slots (percent)

Figure 2: Throughput and collisions comparisons.

Model Count	Non-Vect. Murmur (ns)	Vect. Murmur (ns)	Non-Vect. RMI (ns)	Vect. RMI (ns)
10	2.4	1.9	10	4
10^3	2.4	1.9	13	4
10^5	2.4	1.9	25	8
10^7	2.4	1.9	112	22

Table 1: Run time of vectorized RMI and Murmur.

traditional hash functions and two efficient learned models: RMI [11] and RadixSpline [9].

Run Time. Figure 2(a) shows the median throughput for hashing the sequential dataset with 10% removed elements using traditional hash functions and non-vectorized learned models, while varying the count of line segments used in each model. As expected, traditional functions are much better than non-vectorized learned models, even with small sizes. The throughput of learned models decreases significantly for large sizes due to the increased number of cache misses when accessing the model parameters. Table 1 shows the performance of AMAC-based vectorized versions of both RMI and Murmur hashing for the same dataset. As shown, with 10^3 models, the performance gap between non-vectorized RMI and Murmur is substantial (around 10 ns), however, using the AMAC-based vectorization reduces this gap to be 2 ns only. Note that, at very large models (e.g., 10^7), RMI becomes much slower than Murmur, even with vectorization.

Collisions. Figure 2(b) shows the amount of collisions for RadixSpline against Murmur hashing on various datasets. RMI is omitted as it has similar results to RadixSpline. Here, we mapped N elements into N slots and report the proportion of empty slots. The dashed line is the theoretically expected value for true uniform random hash functions. As shown, for many datasets, learned models indeed have less empty slots than hash functions (i.e., less collisions). However, for *fb* and *osm* datasets, the models make the collisions worse. This confirms our analysis in Section 3.1.

Bucket Chaining Hash Tables. Bucket chaining hash tables deal with collisions by creating linked lists for the keys mapped to the same location. When retrieving a key, we traverse the linked list until we find the key. With increased collisions, the space needed for the chained hash tables increases. Figure 3(a) shows the effect of using different hash functions and RadixSpline as a representative learned model when building bucket chaining hash tables with different payload and bucket sizes. We observe that RadixSpline can lead to less probe times for all of the datasets, except *fb* and *osm* ones. Moreover, larger payloads lead to larger cache miss penalties, and hence with increasing payload sizes, hash functions take slightly more time than learned models.

Cuckoo Hash Tables. Having a high primary key ratio reduces the unnecessary lookups, as one avoids going to the second location, and hence improves the probe time. We show the effect of replacing one of the used multiple hash

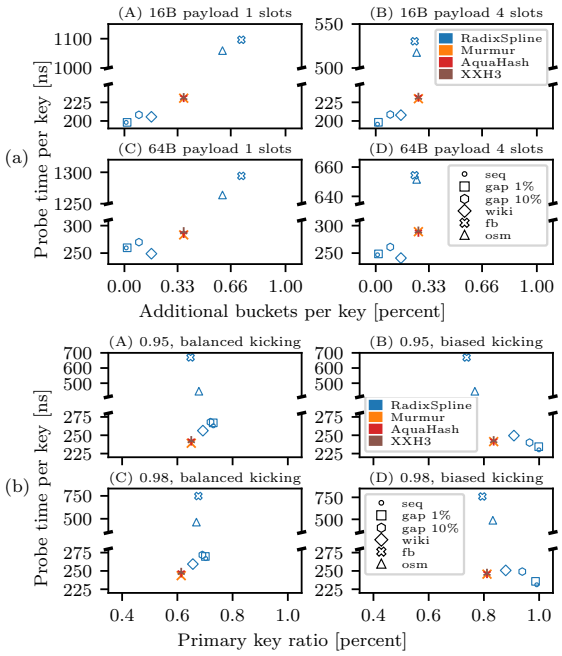
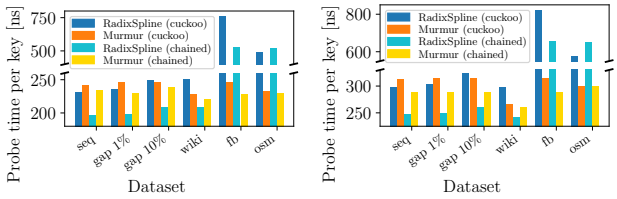


Figure 3: (a) Bucket chaining hash table probe times for varying payload sizes and slots per bucket. (b) Primary key ratio and probe times for various Cuckoo kicking strategies and load factors.



(a) 16 bytes payloads (b) 64 bytes payload

Figure 4: Probe times comparison between Cuckoo and bucket chaining hashing.

functions by a learned model. We use a Cuckoo Hash with 2 hash functions, load factor of 1, bucket size of 8, and two kicking strategies; balanced and biased [8]. As shown in Figure 3(b), using any two traditional hash functions consistently achieves primary key ratios of 62% and 83%, for biased and balanced kicking, respectively, which are close to theoretically optimal. However, we observe that using learned models, e.g., RadixSpline, along with both kicking strategies can lead to better primary key ratio for all datasets, except *fb* and *osm*. With biased kicking, learned models get a much better primary key ratio which leads to lower cache misses and thus a slightly better probe time.

Combined Probe Times. Figure 4 shows the probe times achieved by employing each of RadixSpline and Murmur hashing inside both bucket chaining and Cuckoo hash tables on various datasets. We used bucket size of 4 for both tables. Also, biased kicking is used for constructing the Cuckoo table. As shown, for all datasets except *fb* and *osm*, bucket chaining with RadixSpline is the best strategy. Cuckoo tables are generally slower than their chained counterparts.

5. ACKNOWLEDGEMENTS

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, and NSF IIS 1900933. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. Finally, this research was partially supported by the NSF, under grant #2030859 to the Computing Research Association for the CIFellows Project.

6. REFERENCES

[1] Intel Advanced Encryption Standard (AES) New Instructions Set. `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf`.

[2] Austin Appleby. Murmurhash3 64-bit finalizer. `https://code.google.com/p/smhasher/wiki/MurmurHash3`.

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[4] Yann Collet. xxHash. `https://cyan4973.github.io/xxHash/`.

[5] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[6] Intel Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[7] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.*, 11(11):1702–1714, 2018.

[8] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter A. Boncz, and David G. Andersen. Cuckoo Index: A lightweight secondary index structure. *Proc. VLDB Endow.*, 13(13):3559–3572, 2020.

[9] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. In *Proc. of aiDM@SIGMOD*, 2020.

[10] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.*, 9(4):252–263, 2015.

[11] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, page 489–504, 2018.

[12] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The Case for a Learned Sorting Algorithm. In *SIGMOD*, page 1001–1016, 2020.

[13] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[14] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[15] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proc. VLDB Endow.*, 9(3):96–107, 2015.

Algorithm 1 Function HashViaLearnedModel (Instances s, Keys *input*, KeysNum N, Output *hashes*)

```
1: done ← 0 /* Flag to end hashing computation */
2: state ← InitializeFSMInstances (s) /* Initialize s instances of a finite state machine*/
3: while done < s do
4:     k = (k == s) ? 0 : k
5:     switch state[k].stage do
6:         case P: /* Predict using the root model, and prefetching next model parameters */
7:             if i < N then
8:                 state[k].vkey ← LoadKeyVec(input, i)
9:                 pred ← PredictNextLevelModelIndVec (state[k].vkey, root)
10:                state[k].stage = H
11:                i+ = W /* W is the vector width (Assume N mod W) = 0*/
12:                PrefetchModelParamsVec (pred, state[k].vparams)
13:            else
14:                state[k].stage = D
15:                ++done
16:            end if
17:        case H: /* Perform actual hashing */
18:            hashes ← HashKeysVec (state[k].vkeys, state[k].vparams)
19:            state[k].stage = P /* Initiate prefetching for a new set of keys in P*/
20: end while
```


[16] J. Andrew Rogers. AquaHash. `https://github.com/jandrewrogers/AquaHash/`.

APPENDIX

A. COLLISION ANALYSIS

If we assume that g_i's are generated from an independent and identically distributed (iid) variable with probability density function $f_G(x)$, then the expected number of empty slots e, after mapping keys to their hash outputs, is given by the formula below:

$$\mathbf{E}[e] = N \cdot \int_0^1 (1-x) \cdot f_G(x) dx$$

This equation was derived using the fact that if the gap value x is less than one, then with probability x, a location boundary would fall between the two consecutive values. This is because the location boundaries are separated by unit values and the probability of a random boundary falling in a gap of size x is x. The probability that no boundaries fall in a gap of size x is $(1-x)$ and $(1-x) * f_G(x)dx$ represents the proportion collisions for gap values from $[x, x+dx]$. This quantity is then integrated from 0 to 1, as consecutive keys with gaps beyond one don't collide.

B. ALGORITHM FOR HASHING VIA LEARNED MODELS

Algorithm 1 shows the pseudo code of our proposed batch-oriented hash function that combines AMAC with vectorized learned models. The core idea is sample: for a vector of keys, we map the hashing computation generated by 2-levels learned model into an FSM with two states, where the first state (Lines 6 to 16) uses the root model to predict the index of the second level model, and prefetches its parameters, and the second state (Lines 17 to 19) performs the actual hashing using the prefetched model parameters. The algorithm keeps interleaving multiple running instances of the FSM till it finishes hashing all input keys. The logic in each state is completely implemented with SIMD instructions.