Can Learned Models Replace Hash Functions?

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

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

Tim Kraska
MIT CSAIL
kraska@mit.edu

ABSTRACT

Hashing is a fundamental operation in database management, playing a key role in the implementation of numerous core database data structures and algorithms. Traditional hash functions aim to mimic a function that maps a key to a random value, which can result in collisions, where multiple keys are mapped to the same value. There are many well-known schemes like chaining, probing, and cuckoo hashing to handle collisions. In this work, we aim to study if using learned models instead of traditional hash functions can reduce collisions and whether such a reduction translates to improved performance, particularly for indexing and joins. We show that learned models reduce collisions in some cases, which depend on how the data is distributed. To evaluate the effectiveness of learned models as hash function, we test them with bucket chaining, linear probing, and cuckoo hash tables. We find that learned models can (1) yield a 1.4x lower probe latency, and (2) reduce the non-partitioned hash join runtime with 28% over the next best baseline for certain datasets. On the other hand, if the data distribution is not suitable, we either do not see gains or see worse performance. In summary, we find that learned models can indeed outperform hash functions, but only for certain data distributions.

PVLDB Reference Format:
Ibrahim Sabek, Kapil Vaidya,
Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. Can Learned Models Replace Hash Functions?. PVLDB, 16(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DominikHorn/hashing-benchmark.

1 INTRODUCTION

Hashing and hashing-based algorithms and data structures find countless applications throughout computer science, such as in machine learning, computer graphics, bioinformatics, and compilers (e.g., [13, 42, 52, 56]). Hashing is also a fundamental operation in database management (e.g., [8, 37, 67]), including playing a key role in the implementation of numerous core database data structures and algorithms (e.g., indexes [37, 38], filters [36], joins [8], partitioning [64], and aggregation [26]). Due to its numerous applications, considerable research efforts have focused on introducing efficient hashing functions (e.g., [49, 52, 61, 67]).

Traditionally, hash functions aim to mimic a function that maps a key to a random value in a specified output range. For typical cases where the size of the output range is linear in the number of keys, this random assignment results in colliding keys. A collision occurs when multiple keys get mapped to the same output value. A typical hash index approach allocates a number of fixed size slots (the number of slots generally being a constant times the expected number of keys) and maps incoming keys into these slots using a hash function. The ideal case for indexes would have no keys collide, so each key goes to its own separate slot. This would make key lookups and updates faster, as one would simply check the corresponding slot for the key. With truly random hash functions, collisions are unavoidable, and one can readily calculate the expected number of collisions given the number of slots and keys [55].

Naturally, there are many well-known schemes like chaining, probing, and cuckoo hashing to handle collisions. As the name suggests, chaining handles collisions by creating a chain of colliding keys. Probing checks neighboring slots to find an empty slot to place the key. Cuckoo hashing handles collisions by using multiple hash functions to provide alternate slots for colliding keys. For each of these schemes, more collisions reduces their performance.

Another approach to build hash indexes is to use *perfect* hash functions instead of truly random hash functions. Perfect hash functions have no collisions; however, they must be specially constructed for a given dataset, and have other costs in storage and computation time.

In recent years, several works have utilized the idea of using machine learning to improve the performance of many database components (e.g., [39, 54, 70]) and basic data structures (e.g., [24, 25, 43, 50]). By using machine learning to explicitly capture trends in the underlying data, these methods can aim for instance-optimal performance. For example, in a recent benchmarking study [53], it has been shown that learned index structures (e.g., RMI [43], RadixSpline [40]), which employ CDF-based learned models, can outperform traditional indexes on practical workloads.

As one direction in this line of research, it was suggested in [43] that such learned models can be used to obtain an efficient hash function with fewer collisions. They also provided some empirical evidence that a hash index with learned model as the hash function can have better performance than using a truly random hash function. What is unclear, however, is when such learned models are effective in replacing existing hash functions in applications. At one end, *traditional* hash functions [29, 81] are fast to compute, but suffer

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

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

from collisions [78] that can reduce query performance. On the other hand, *perfect* hash functions [52] avoid collisions, but are difficult to construct [46], and are not scalable [21], in the sense that the size of the function representation grows with the size of the input data. As an alternative, learned models can potentially provide a better tradeoff between computation and collisions. If the model learns a good approximation of the empirical CDF of the input keys, we may achieve few collisions; and if the data allows a compact learned model, we may achieve a model size independent of or very slowly growing with the input data size.

Surprisingly, though, we are not aware of a thorough experimental study examining the learned models against both traditional and perfect hashing in query processing operations like indexing and joins. We aim to remedy that here. We make the following contributions:

- We provide an analysis of the factors affecting collisions for learned models, helping us to identify situations where they can have fewer collisions than traditional hash functions.
- We perform an extensive benchmarking study for traditional, perfect, and learned model based hash functions. We benchmark them through three different applications: hash table probing/inserting, range querying, and joins. We test using multiple synthetic and real-world datasets.
- Through the empirical study and analysis we find useful insights on when to use learned models instead of traditional and perfect hashing in various database applications.
- We provide a unified open-source implementation for the baselines used in our experiments[1].

In summary, our collisions analysis and experimental benchmarking demonstrate that, for datasets with a well-ordered distribution of gaps between their keys, learned models can result in lower collisions than traditional hash functions. For these datasets, using learned models can improve the probe and insert throughputs of hash tables. Such improvement varies with the hashing scheme (strongest with bucket chaining, and weakest with cuckoo hashing), the load factor, and the bucket capacity. In many other cases, however, such as with data from typical distributions (e.g., normal) or having string keys, we do not see collision reduction using learned models. We also find that using learned models with bucket chaining can support range queries, and provides the best throughput in mixed workloads (point and range queries) that have a majority of point queries. Finally, we find that using learned models in non-partitioned hash join [45, 80] results in improved performance for favourable datasets.

2 TRADITIONAL HASH FUNCTIONS

A uniform hash function $h(x) : X \mapsto U$ attempts to map arbitrary inputs to independent and identically distributed (i.i.d.) uniform random outputs. Obtaining true randomness is not feasible in practice [42]. However, state-of-the-art hash functions appear to come reasonably close to imitating true randomness in many practical settings [60, 81]. The extent to which a hash function avoids collisions, i.e., instances where two distinct inputs map to the same output, is often referred to as the its' quality. There is a seemingly endless supply of different proposed hash functions to choose from [81]. Here, we briefly give a background on some of the well-known functions that we study in the paper.

[1]https://github.com/DominikHorn/hashing-benchmark

Multiplicative Hashing (MultiplyPrime). This method is prominently described by Donald Knuth [42] as a family of hash functions with great properties for practical applications. He explicitly advertises their non-uniform random properties, i.e., sensitivity to the data distribution, as a strength [42]. Let A be a constant, relatively prime 2^w with w being the machine word size. Then, the following function produces outputs in $[0,M)$.

$$h(x) = \left\lfloor M \cdot \left(\left(\frac{A}{2^w} x \right) \mod 1 \right) \right\rfloor$$

The trick to make this efficient is to avoid fractional computations by shifting the calculation by w, i.e., to multiply with $\frac{A}{2^w} \ll w = A$ instead of the complex decimal computation: $h(x) = \lfloor \frac{M}{2^w} \cdot (Ax \mod 2^w) \rfloor$. Neatly, this gets rid of the modulo since most physical machines with a word size w will naturally compute everything $\mod 2^w$. According to Knuth, M should be some power of the machine's radix [42] to ensure that we are including the more significant bits in the final result.

Fibonacci Hashing (FibonacciPrime). It is an instance of multiplicative hashing, choosing $\frac{A}{w} = \Phi^{-1} = \left(\sqrt{5}-1\right)/2$ based on the golden ratio. It promises to inherit Φ^{-1}'s neat scattering characteristics, i.e., that each added consecutive element falls in the largest remaining interval, dividing it by the golden ratio [6, 42, 75, 76, 83]. As in multiplicative hashing, we implement Fibonacci hashing using the integer multiplication trick. However, this time we choose $C = \Phi \cdot 2^w$ with w as the machine word size. Some implementations also round C to the next closest prime.

Murmur Hashing (Murmur). It is a family of simple and fast hash functions developed by Austin Appleby [3, 4], and has been studied extensively in previous works (e.g., [2, 67]). Its name is derived from the original idea for its implementation, i.e., repeatedly applying **mu**ltiply and **r**otate instructions to imitate true randomness. However, it ended up being implemented as a sequence of multiply, shift, and xor operations. In particular, its 64-bits finalizer merely consists of three xors, three shifts, and two multiplications [3, 67].

XXHash. It is a widely used open-source uniform hash function with support for many programming languages [18]. It targets RAM speed limits for hashing large enough blobs of data, all while promising decent performance on small inputs.

AquaHash. It is a uniform hash function that utilizes Advanced Encryption Standard (AES) intrinsics [33], i.e., AES encryption primitives implemented in hardware on many modern CPUs [69]. In a previous study, AquaHash has shown promising results compared to XXHash and Murmur for small keys [71].

3 LEARNED MODELS AS HASH FUNCTIONS

Learned index structures [43] approximate the cumulative distribution function (CDF) of the data to predict the position of a lookup key in a sorted array. When the data has a learnable pattern, i.e., has low entropy, learned indexes can be much smaller than the input data itself. While initial proposals considered using neural networks to approximate the CDF, state-of-the-art learned indexes use a collection of simple *linear models*, which we refer to as submodels; these are fast to both learn and evaluate. Some indexes aim to minimize the root-mean-squared-error (i.e., L2 loss) [43] and others bound the maximum prediction error. Assuming a perfect modeling of the CDF, a learned index would constitute a perfect order-preserving hash function, i.e., a collision-free mapping from keys to positions. For the rest of this paper, we refer to Learned Model based Hash functions as *LMH*. Since real-world data contains many irregularities that make it hard to approximate, a learned index inevitably needs to trade off precision for space. With larger models, inference time increases because of limited cache sizes [53]. We describe the three main learned indexes we evaluate for hashing.

Recursive Model Indexes (RMI). The RMI index is a multi-stage model combining simpler models [43]. When the data fits into memory, an RMI rarely has more than two stages. It is built in a "top-down" fashion. The stage-one model computes a rough approximation of the CDF, which is scaled between 0 and the branching factor B. This value is used to select a second-stage model, which approximates the local distribution of the data and is used to produce the final approximation. In other words, the stage-one model partitions the data into B buckets and each second-stage model approximates the data that falls into its corresponding bucket. A recent study [53] showed that RMI, amongst other indexes, achieves the best tradeoff between inference time and space.

Radix Spline Indexes (RadixSpline). It is another learned index variant [40], that is built "bottom up", and consists of a linear spline [59] to approximate the CDF and a radix lookup table that indexes resulting spline points. Compared to RMI, RadixSpline can be built in a single pass with constant cost per element. RadixSpline's spline-building algorithm [59] bounds the maximum prediction error. Besides the maximum error, RadixSpline is parameterized with a certain number of radix bits that define the size of the radix table. Lookups first consult the radix table, which indexes r-bit prefixes of spline points and is used to narrow the search range over the spline points. Then binary search is used on the narrowed range to identify the two spline points surrounding the lookup key. Finally, linear interpolation between the two spline points is used to obtain a prediction. The necessity to search over the spline points make it somewhat slower than RMI which does not require any search in inner nodes.

Piece-wise Geometric Model Indexes (PGM). Similar to RadixSpline, the PGM index [30] provides an error-bounded approximation of the CDF. It consists of multiple levels where each level represents an error-bounded piece-wise linear regression (PLR). In contrast to a spline where consecutive spline points are connected, a PLR additionally stores an intercept value with each point. Like RadixSpline, PGM is built "bottom up" but instead of using a radix layer it recursively applies its PLR algorithm until a certain error threshold has been met. PGM can also be built in a single pass with constant amortized cost per element. Due to its multi-level structure, PGM can have slightly higher inference cost than RadixSpline [53] but is more robust when outliers are present. Note that we explore static PGM only in our study.

4 PERFECT HASHING

Where traditional hash functions aim to produce (near)-i.i.d. uniform random outputs, *perfect hash functions* provide an injective function that maps a set of elements into a range. That is, for a given input set, the function will produce no collisions [10, 28, 32, 52]. Here, we focus on two types of perfect hash functions: *minimal perfect* (MPHF), and *order preserving minimal perfect* (OMPHF). We first explain the corresponding definitions, and then describe the state-of-the-art MPHF and OMPHF algorithms we study.

Perfect. A hash function $h(x): X \mapsto [0,N]$ is *perfect* for the domain X if it is injective. Equivalently, it produces zero collisions in the output domain ($\forall x_1,x_2 \in X : h(x_1)=h(x_2) \implies x_1=x_2$).

Minimal. A hash function $h(x): X \mapsto [0,N]$ is *minimal perfect* if it is perfect and a bijection; that is, each element of the output range has a single corresponding domain element (Perfect, and additionally $\forall y \in [0,N] : \exists x \in X \mid h(x) = y$). The information theoretical lower bound for storing a minimal perfect hash function is $\lg e \approx 1.44$ bits per key [10, 15, 28, 32, 35] since key-related information is not retained after construction. For this reason, querying with non-keys (unknown keys) generally yields arbitrary results.

Order Preserving. Order preserving perfect hash functions order their outputs according to the original relative order $\leq$ of input elements ($\forall x_1,x_2 \in X : x_1 \leq x_2 \implies h(x_1) \leq h(x_2)$). Being able to store any arbitrary data order induces an $\Omega(n \log n)$ space cost [10].

Comparison to Traditional Hashing. In general, building a MPHF, $h(x): X \mapsto [0,N]$, requires knowing the entire input set X a priori. In many implementations, the set X is not stored or reconstructible after the MPHF is built. Querying with a non-key $x' \notin X$ generally yields some arbitrary output value; most often, $h(x') \in [0,N]$, but this is not guaranteed. MPHF are generally not easily updated in place; often a full rebuild is performed if a new element is inserted, or other expensive (non-constant) time work. Compared to traditional hashing, where only constant work is necessary for initialization, MPHF and OMPHF generally require running a one time $O(n)$ build algorithm before they can be used.

Recursive Splitting (RecSplit). It is a MPHF which has been shown to deliver state-of-the-art results in regards to space usage, lookup, and build time [28]. Specifically, it comes close to achieving the theoretically optimal 1.44 bits per key in practice, while only requiring expected linear and constant times for construction and lookups, respectively [28]. RecSplit works by recursively partitioning inputs into ever smaller buckets until brute force search for a MPHF, i.e., a *bijection*, is viable. The threshold for this search, called *leaf size* l, as well as the average *bucket size* b for partitioning are parameters of the construction algorithm. RecSplit utilizes an indexed family of uniform random hash functions (examples in Section 2). This enables efficiently encoding the tree of brute-force determined indexes using an optimal Golomb-Rice instantaneous code [28, 72].

MWHC. It was originally proposed as a family of OMPHFs with expected $O(n)$ construction and $O(1)$ access time [52]. It has been extended to provide a practical MPHF with constant access and requiring 3 bits per key storage [10, 14]. We refer to our simplified implementation of the latter approach as *BitMWHC*. Abstractly, MWHC utilizes a hypergraph to efficiently find a solution for a randomly generated system of linear equations that is used to store the desired order preserving hash function $f(x): X \mapsto U$ given by $f(x) = v(h_1(x)) \diamond \ldots \diamond v(h_k(x))$, where each h_i denotes a distinct uniform random hash function, $v(x)$ maps each hash function output to a value in U and $\diamond$ reduces $U \times U$ to U. In practice, $v(x)$ may, for example, be implemented as a simple array of values, h_i as a family of reasonably high quality hash functions such as Murmur with seed values, and $\diamond$ as *xor* or as *addition* with an additional modulo computation at the end.

The construction algorithm first builds a k-hypergraph with each of the $\lambda|X|$ vertices corresponding to one entry of $v(x)$ and one edge

$[h_1(x),...,h_k(x)]$ for each input, where k and λ are user-defined parameters. All h_i are randomly chosen from a suitable family of hash functions as described above. A valid assignment for $v(x)$ exists, i.e., $f(x)$ is solvable iff this hypergraph is acyclic. A simple peeling scheme is used to both determine acyclicity and the order in which we can safely assign values to each $v(x)$ to yield the desired values for $f(x)$ for each $x \in X$. We simply restart if the acyclicity test fails, hence the *expected* $O(n)$ construction time [52]. For $k=3$, we require $\lambda \geq 1.23$ to efficiently find a suitable acyclic hypergraph [52, 57].

5 HASHING SCHEMES

When collisions occur in a hash table, they are resolved using *hashing schemes*. In this section, we give a brief background on the hashing schemes we study in this paper. In each scheme, we discuss how the hash table is implemented and how collisions are handled.

5.1 Bucket Chaining (CHAIN)

Bucket chaining is a classic collision resolving scheme [8, 67, 71]. In this scheme, the hash table is implemented as an array of pre-allocated buckets, where each bucket stores multiple tuples, with collided keys, at a specific slot in the table. To insert a tuple, the key of this tuple is first hashed to a slot in the hash table, and then the whole tuple is first tried to be placed in the corresponding bucket at this slot. If the current bucket is already filled up, a new one is created, pre-allocated and chained to it. To query for a tuple, the query key is first hashed to a slot in the table (similar to what happens in inserts), then the chain of buckets at this slot is traversed until either the matching tuple is found or the end of the chain is reached (i.e., the matching tuple is not found). In general, bucketization improves the data locality, and reduces the number of cache misses. That being said, choosing the bucket size should be carefully tuned to avoid wasting large spaces.

5.2 Open-Addressing

In open-addressing, all tuples are inserted in the hash table slots themselves, without extra chains to handle collisions. In case of a tuple with a colliding key, the hash table slots are probed (i.e., searched), until a slot is found to place the tuple [19, 67]. Typically, a *probing scheme* decides the set of hash table slots to check, referred to as a probing sequence, till a place is found to insert the tuple. Query operations follow the same probe sequence. There are two main categories of probing schemes: (1) schemes that probe for the first available (i.e., empty) slot, and (2) schemes that evict the existing tuple at the probe location (i.e., when a collision occurs) and replace it with the new tuple. In this paper, we study an example of each of these two categories (linear probing and cuckoo hashing).

Linear Probing (LP). This is the most basic probing scheme for collision handling in open-addressing. In this scheme, when inserting (or querying) a tuple, the key of this tuple is first hashed to obtain a hash table slot (i.e., initial probe location). Then, the hash table is sequentially traversed starting from this slot. In case of insertion, the traversal stops if an available slot is found. In case of querying, the traversal stops if we find either the matching tuple or an empty slot (i.e., matching tuple is not found). Linear probing has two main advantages: (1) its simple design, and (2) cache efficiency due to the sequential scan. In contrast, its performance degrades when large contiguous blocks of hash table slots are occupied, referred to as primary clusters. In this case, the number of nearby empty slots around each probe location is significantly reduced, and the scheme tends to have long probe sequences. Such performance issue can be avoided by either (1) increasing the hash table size such that the percentage of its occupied slots (a.k.a load factor) is always kept less than 60% [67] or (2) carefully tuning its update operations [11]. We note that there are two other popular variants of linear probing: (1) quadratic [19, 42], and (2) robinhood [17], which are efficient for write-heavy and high unsuccessful lookup workloads, respectively. However, according to a recent study [67], linear probing outperforms both of them using the appropriate load factor. Therefore, we focus on linear probing here.

Cuckoo Hashing (CUCKOO). Cuckoo hashing [61] provides another useful alternative hash table design. A simple variation of cuckoo hashing uses two subtables, where each subtable has an independent hash function. To insert a tuple, the key of this tuple is hashed with the first (or primary) hash function to obtain a slot in the primary table. If this primary slot is available, then the tuple is inserted and the probe sequence ends. Otherwise, the tuple tries to be inserted in the second (or secondary) subtable using the second hash function. If the secondary slot is occupied as well, then a kicking strategy is applied to evict the existing tuple in either the primary or the secondary slot, and replace it with the current input tuple. After that, the evicted tuple is reinserted again, following the same steps. The eviction chain continues until either all evicted tuples are successfully inserted or a maximum chain length is reached. This last case is a failure; one solution is for all tuples in both hash tables to be rehashed with two new hash functions.

With balanced kicking [61], the primary or the secondary slot is randomly selected for eviction. In biased kicking [22, 38], the tuple in the secondary slot is preferred for eviction, which has been shown to improve performance for positive lookups. We experimentally found that biased kicking performs better, so we use it throughout all our experiments involving cuckoo hashing.

To probe for a tuple, we need only to check the primary and secondary slots, which yields at most two cache misses regardless of the load factor. However, a major drawback of the simple variation of cuckoo hashing is the failure case, where the maximum length of the eviction chain is reached, happens at low loads. Higher loads can be handled by generalizing to use more hash tables (e.g., 4 instead of 2) [31, 67] or allowing multiple tuples per slot [2, 23, 71]. In this paper, we employ the bucketized variant, where each hash table slot allows more than one tuple, which again limits to two cache misses when a bucket fits in a cache line.

6 COLLISIONS ANALYSIS FOR HASHING

Here, we identify and analyze the factors affecting collisions for both *LMH* and traditional hash functions. This analysis helps us to identify situations where *LMH* can have fewer collisions than traditional hash functions. We specifically focus on *LMH* functions with piece-wise linear submodels for this analysis.

Notation. For ease of analysis, we start by focusing on the task of mapping N keys to N locations. This analysis readily generalizes, and the high-level conclusions are independent of this assumption, with the main difference being the number of locations increases, the number of collisions decreases. Assume that we apply a hash function f on the keys, where f could be a traditional hash (Section 2) or a *LMH* function (Section 3). Let $x_0,x_1,...,x_{N-1}$ be the sorted array of the N input keys, and let $y_0,y_1,...,y_{N-1}$ be the sorted array of the hashing outputs $f(x_0), f(x_1),...,f(x_{N-1})$ (note that $y_i = f(x_j)$ for some j, but y_i is not necessarily $f(x_i)$). For *LMH* functions, the y_i's may be on the real-valued range $[0,N)$, and we would then map each key to the location corresponding to the value of y_i rounded down to an integer. For convenience, we let $y_{-1}=0$. The sorted output values generate a set of gaps $g_0,g_1,g_2,...$ such that $y_i = \left(\sum_{t=0}^{i} g_t\right)$. We assume that g_i's are i.i.d, with probability density function $f_G(z)$ and CDF $F_G(z)$; this is a reasonable approximation for analysis. For example, for uniformly randomly distributed outputs $f(x_i)$, the gaps between y_i are approximately exponentially distributed [55].

Characterizing Collisions. A collision occurs when two keys are mapped to the same location. The key insight regarding collisions is that they depend on the gaps between consecutive sorted hashing output values ($y_i - y_{i-1}$). If the gap between two consecutive values is greater than one (i.e., $y_i - y_{i-1} \geq 1$), then the corresponding keys would definitely be placed in separate locations. On the other hand, if the gap is smaller than one (i.e., $y_i - y_{i-1} \leq 1$), the corresponding keys may be mapped to the same location; it depends where y_i and y_{i-1} relative to the integer boundary.

Ideally, we would want all the gaps to be more than one, to have zero collisions. However, the gap values are constrained by the condition that the sum of all the gaps should be less than the number of locations which is N here.[2] Thus, the gap distribution would have to be the trivial distribution that is always 1 to avoid collisions.

Let c be the number of colliding keys (i.e., keys that are not alone in a location). Assuming that f is not a lattice distribution[3], we can describe the expected number of colliding keys $\mathbf{E}[c]$ with the following lemma. In the below, recall $\{x\} = x - \lfloor x \rfloor$.

Lemma 1. *As N grows large, $\mathbf{E}[c]$ converges to*

$$N\left(1-\int_{u=0}^{1}\left(\int_{t=1-u}^{\infty}(1-F_G(1-\{t+u\}))\cdot f_G(t)dt\right)du\right).$$

We remark that the proof reveals that this formula is also a good approximation for large N.

Proof. Let Z_i be the indicator random variable that is 1 if y_i is alone in its own location. We first consider the position of y_{i-1}. For sufficiently large i, $\{y_{i-1}\}$, the fractional part of y_{i-1}, is known to converge to the uniform distribution on $[0,1]$ (see, e.g., Thm 5.8.4. of [41]). We therefore treat $\{y_{i-1}\}$ as being distributed uniformly on $[0,1]$. Accordingly, the probability y_i is in a different location from y_{i-1} is given by

$$\int_{u=0}^{1}\left(\int_{t=1-u}^{\infty} f_G(t)dt\right)du.$$

We also need, however, that y_{i+1} is also in a different location from y_i. This depends on the value of $\{y_i\}$. Taking this into consideration yields the following probability for Z_i:

$$Pr(Z_i=1) = \int_{u=0}^{1}\left(\int_{t=1-u}^{\infty}(1-F(1-\{t+u\}))\cdot f_G(t)dt\right)du.$$

As N grows large, the approximation of uniformly distributed $\{y_{i-1}\}$ is arbitrarily accurate for almost all i, giving the convergence. □

[2]Sum of gaps is: $\sum_{t=1}^{N-1}(y_t - y_{t-1}) = y_{N-1} - y_0 \leq N$.
[3]Lattice Distribution: A discrete probability distribution concentrated on a set of points of the form a+nh, where h>0, a is a real number and n=0,±1,±2,.

Collisions for Traditional Hash Functions. In case of a truly random hash function, the output values will be uniformly distributed in the range $[0,N]$ irrespective of the input distribution. Therefore, the gap distribution of the output values is very well approximated by the exponential distribution with mean 1. Most traditional hash function displayed this behaviour in our evaluation.

Collisions for *LMH* Functions with Piece-wise Linear Submodels. To gain intuition, let us start by using a *single* linear model to approximate the CDF of the input data $x_0,x_1,...$, and this will give us our hash function f. Let the linear model be $m*(x-x_0)$ where m is $(N-1)/(x_{N-1}-x_0)$. Note that the slope would be approximately the mean of the gap distribution of the input keys. The resulting hash function would be $h(x) = m*(x-x_0)$ which maps the input keys in the range $[0,N)$. After applying this hash function to obtain the output values $y_0,y_1,...$, we notice that the gaps between the output values are simply the scaled version of the gaps between the input keys: $y_{i+1}-y_i = (x_{i+1}-x_i)*m$. At a high level, if the input is evenly spaced, then our outputs will similarly be evenly spaced, resulting in fewer collisions. If the input gaps are high in variance, we would expect more collisions. In *LMH* functions, this scaling would happen at the submodels scale.

Accordingly, if the data is generated similarly to our theoretical model, with a gap distribution g' $(x_0, x_1 = x_0 + g'_0, x_2 = x_1 + g'_1, ...)$, the gap distribution of the input keys determines the gap distribution of the output keys and thus the amount of collisions. In certain cases, like auto-generated keys (1,2,3,4,5,...) perhaps with some deletions or noise, the input gaps are mostly constant. In this scenario, a piece-wise linear model can lead to fewer collisions than a traditional hash function. However, if the input keys are generated by sampling from a distribution instead of sequentially, multiplying the CDF value of the key by the array size will behave as an order-preserving hash function. A *LMH* function that approximates this underlying distribution would behave essentially the same as a truly random hash function in terms of collisions.

Increasing the number of submodels can improve the accuracy of when using a piece-wise linear model to approximate a CDF. This helps in the case of indexing an item, but from our argument above, we see that this does not necessarily reduce the number of collisions. We show this via an example. We mapped 100 million uniform randomly and normally distributed keys to 100 million slots using RMI with varying number of submodels. In Figure 1, we plot the proportion of collisions as we increase the number of submodels in RMI. We observe that for uniform randomly distributed keys increasing the number of linear submodels does not affect collision metric until we reach 50 million submodels. RMIs with 100 submodels and 100000 submodels are both able to approximate the CDF of the distribution well and the output is approximately uniformly randomly distributed in both cases. The larger RMI provides better accuracy than the smaller one but essentially the same number of collisions. The RMI with 50 million submodels essentially memorizes the empirical CDF of the dataset and thereby results in lower collisions. For the normal distribution, an initial increase in the number of submodels reduces collisions as an RMI with only 1-2 submodels fails to approximate the CDF of normal distribution well.

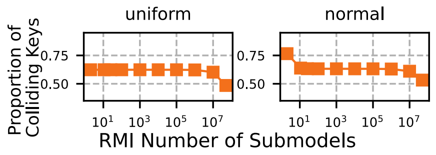

Figure 1: Proportion of collisions with increasing RMI size.

	wiki	*fb*	*osm*	*book*	*gap_10*	*uniform*	norm	lognorm
RMI	10^3	10^7	10^7	10^6	10	10^2	10^2	10^4
RadixSpline	10^3	10^8	10^8	10^7	10	10^2	10^2	10^4

Table 1: Default numbers of submodels in *LMH* functions.

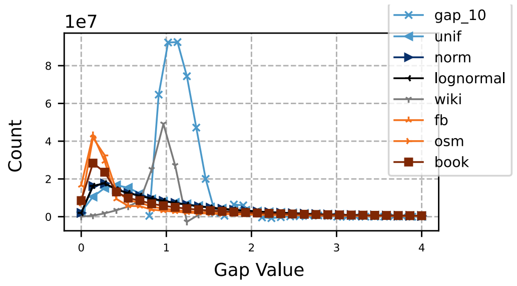

Figure 2: Gap distribution of various datasets

7 EVALUATION

In this section, we present an empirical study for the performance of *LMH* functions and compare them against both traditional and perfect hashing. Our main objective is to answer the following question: *what are the main workload characteristics, scenarios, and operations where employing LMH functions would improve performance?* We first study the collisions and computation time tradeoffs (Section 7.2). Then, we evaluate the performance of the various types of hash functions in supporting the main hash table operations, lookup and insertion, for different types of hash tables (Section 7.3). We also provide more detailed experiments regarding issues such as how collisions, key types, and payload size affect performance in practice, as well as the impact of construction time for *LMH* (Section 7.4). Finally, we move to some higher-level operations that use hash tables, and show cases where *LMH* can improve the performance of range queries (Section 7.5) and non-partitioned hash join (Section 7.6).

7.1 Experimental Setup

Datasets. We use both real and synthetic key datasets in our experiments. All keys are 64-bit integers[4]. For real keys, we use the four datasets from the SOSD benchmark [53]. These datasets are (1) *fb*, which has randomly sampled Facebook user IDs, (2) *wiki*, which has timestamps of edits from Wikipedia, (3) *osm*, which has cell IDs from Open Street Map, and (4) *book*, which has keys representing the popularity of books from Amazon. Each dataset has 200 million keys. In any experiment, we use either the whole dataset or a sample from it (details are mentioned in each experiment separately).

For synthetic keys, we use four different key generation processes: (1) *gap_10*, in which sequential keys are first generated at regular intervals of 10 and then 10% of the keys are uniformly randomly deleted (this represents the case of auto-generated IDs after removal of certain users), (2) *uniform*, in which keys are generated uniformly at random in the range $[0,2^{50}]$, (3) *normal* and (4) *lognormal*, in which keys are generated from normal ($\mu = 100$ and $\sigma = 20$) and lognormal ($\mu = 0$ and $\sigma = 1$) distributions, respectively, and then are linearly scaled to the range $[0,2^{50}]$.

As discussed in Section 6, the gaps between sorted hash outputs determine collisions. In order to understand the distribution of these gaps in our datasets, we use an RMI, with 1 million submodels, to map 100 million keys from each dataset to 100 million slots and then plot the gaps between consecutive sorted output values. In Figure 2, x-axis shows the gap value and y-axis shows the count of this gap for some of the used datasets. We observe that *gap_10* and *wiki* datasets have gaps concentrated around 1. *uniform*, *normal*, and *lognormal* datasets have very similar gap distributions concentrated around 0.25-0.35, while *fb*, *osm*, and *book* datasets have significant counts of gaps concentrated around 0.1 (*fb* and *osm* have higher counts than *book*).

In all hash table, range query, and join experiments, we generate 8-byte payloads chosen randomly from the range $[0,2^{64}]$[5]. All tuples (or keys) are randomly shuffled before running any experiment.

Hardware. All experiments are conducted in the main memory on a machine with 256 GB of RAM and an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with Skylake micro-architecture (SKX) and L3 cache of 55MiB. The operating system is Arch Linux with a page size of 4KB (default page size). The implementation of all hashing functions and schemes is our own and in C++. The binaries are compiled with clang++ (12.0.1) using optimization -O3. We have activated prefetching.

Default Settings. Unless otherwise mentioned, we set the number of submodels in RMI and RadixSpline as stated in Table 1. Each value represents the least number of submodels needed to give the least amount of collisions in a specific dataset. For PGM models, we set the error bound to 10. The number of tuples (or keys) in each synthetic dataset is set to 100 million. We use a default bucket size of 1 in bucket chaining. To support cuckoo hashing with a load factor up to 90%, we use a bucket size of 4 as described in [2]. As mentioned in Section 5, we use the biased kicking strategy as it performs better than the balanced one. We set 50000 as a maximum number of kicks. This value led to a suitably small number of insert failures.

Metrics. Throughput is the default metric in most of the experiments. When studying the hash function itself, we use the *computation throughput*, which is the number of hash function operations executed per second. In the hash table and range query experiments, we use the number of completed queries (e.g., probe/insert queries on hash tables) per second (i.e., *queries throughput*). For the join experiments, we use the runtime instead of the throughput to perform a breakdown for the join phases.

Measurement and Profiling. For all experiments, we report the average of three independent runs, where we use a different random seed for generating and shuffling synthetic and real data, respectively, in each run. We use the PerfEvent library [51] to profile the low-level hardware counters in Section 7.3. These counters include L1 and LLC cache misses, branch misses and cycles.

Beyond Scope. Our study focuses only on the single-threaded setup to fairly compare the performance of *LMH* functions with traditional

[4]We focus on integer keys in our study. However, for completeness, we provide a single experiment in Section 7.4 to investigate the performance with string keys.

[5]We focus on 8-byte payloads in our study. However, for completeness, we provide a single experiment in Section 7.4 to investigate the effect of varying the payload size.

and perfect hashing, without parallelism optimizations. That being said, we believe that multi-threaded implementations of these hashing schemes should be evaluated in a standalone study, which we currently plan as an extension for this work.

7.2 Computation Throughput vs Collisions

In this experiment, we are interested in studying the tradeoff between the hash function quality and its efficiency. We use the eleven hash functions previously discussed and five from our datasets[6]. In each dataset, we map a randomly-selected 100 million keys into 100 million hash table slots, and measure both the hash function computation throughput, and the proportion of colliding keys.

Figure 3 shows the results of this experiment. Note that each traditional and perfect hash function is represented as a single point in the scatter plot. However, in *LMH* functions, we vary (1) the number of submodels in RMI and RadixSpline from 1 to 50 million and (2) the error bound of PGM from 1 to 10000, yielding multiple points on the plot. As expected, traditional hash functions have a significant number of collisions, and perfect hash functions are slow. All traditional functions have similar throughput (90-100 million operations/sec) and colliding keys proportion (0.63-0.65) across all datasets. This proportion of colliding keys nearly matches that for truly random hash function which is approximately $(1-1/e \approx 0.632)$. All perfect hash functions have no collisions (by definition), but low throughput (10-20 million operations/sec) due to the high computation overhead coming from either an expensive traversal over the splitting tree in RecSplit [28] or multiple random accesses to the array storing the hypergraph-related values in MWHC [52].

The performance of *LMH* functions, however, depends on the gap distribution of the input datasets as discussed in Section 6. The RMI and RadixSpline hash functions, at their best configurations, can achieve low collisions (0.2 and 0.3) and high throughput (80 to 120 million operations/sec) in two datasets, *gap_10* and *wiki*. For these datasets, the gaps are more or less evenly spaced, and hence *LMH* functions yield a very low number of collisions. In addition, the number of submodels needed for these datasets is small, which makes the *LMH* computation overhead efficient. For *fb*, the variance in the gap distribution is very high, yielding a large number of collisions. Reducing these collisions requires using a large number of submodels (the best proportion of colliding keys is 0.5), yielding low throughput.

In the case of *uniform* and *normal* datasets, we observe that *LMH* and traditional functions have similar collision behavior, regardless of the used number of submodels. This matches our understanding that the CDF-based hashing of *LMH* for these datasets will lead to a distribution of items in buckets that is nearly the same as traditional hashing (as described in Section 6). In general, as discussed in Section 6, increasing the number of submodels in *LMH* functions does not necessarily decrease the collisions. For example, in *wiki*, the proportion of colliding keys using RMI significantly drops from 0.9 to 0.3 after an initial increase in the number of submodels from 1 to 1000, and then becomes stable regardless the number of submodels used.

For the rest of our experiments, we compare *LMH* functions with the best traditional and perfect hash functions, in terms of both computation time and collisions: Murmur and MultiplyPrime for traditional hashing, and MWHC for perfect hashing.

[6]Tradeoffs in *osm* and *book* are similar to *fb*, and in *lognormal* are similar to *normal*.

7.3 Hash Table Performance

Here, we are interested in studying the performance of two main hash table operations; probe and insert.

Probe Throughput. In this experiment, we first insert 100 million tuples in a hash table with varying number of slots (i.e., buckets). Then, we probe the hash table with all the inserted tuples (i.e., query workload), after randomly shuffling them, and measure the throughput. We generate different load factors by varying the number of slots. Figure 4 shows the results for this experiment while using seven input datasets (*uniform* and *normal* nearly have the same results). For each hashing scheme, we use a different range of load factors that are suitable for the scheme. For example, we use load factors $\geq 100\%$ in bucket chaining as it can support inserting tuples more than the total slots in a hash table. Also, we only use high load factors ($\geq 75\%$) with cuckoo hashing because, in smaller load factors, cuckoo hashing is always dominated by other schemes [67].

For bucket chaining, RMI has the best throughput in *gap_10*, *normal*, *lognormal*, and *wiki* datasets, averaging 1.4x better throughput than the second best function, whether it is MultiplyPrime or RadixSpline. This is because RMI has the fewest collisions in these four datasets. Fewer collisions result in shorter chains that need to be traversed during the probe queries, and hence fewer cache misses. In addition, both PGM and MWHC have the worst throughput in *gap_10* and *wiki* datasets due to their high computation overhead[7]. For RadixSpline, we observe an interesting variance in performance in these datasets. It is competitive with RMI throughput in non-skewed datasets (*gap_10* and *wiki*), but becomes the worst in the skewed datasets (*normal* and *lognormal*). This is because in skewed datasets outlier keys lead to having a large radix table with a majority of its entries being useless (i.e., more data structure overhead and cache misses during lookups). This weakness of RadixSpline has been noted in [53]. In *fb*, *osm*, and *book* datasets, we observe a clear ranking among the different hash functions. Although MWHC still has the highest computation overhead, *LMH* functions become the worst options (except *book* in which RMI is slightly better than MWHC) with an average throughput of only 2.5 million queries/sec. This is because of the high number of collisions for *LMH* functions when used with these datasets that have high variance in their gaps distribution.

We also look at the throughput across different load factors. Increasing the load factor increases collisions because there are fewer slots, which degrades the throughput. For example, Murmur has throughputs of 16 and 8.5 million queries/sec at load factors of 25% and 200%, respectively. However, we observe two exceptions to this throughput trend when using: (1) RMI and RadixSpline at load factors between 25% and 100% in *gap_10* and *wiki*, where the throughput actually increases, and (2) MWHC in all load factors, where the throughput is fixed around 8 million queries/sec, regardless of the dataset. The reason for the first exception is that collisions are already close to zero in these two datasets, so increasing the load factor from 25% to 100% primarily reduces empty hash table slots, leading to better caching behavior. The reason for the second exception is that the MWHC computation overhead for each tuple is constant [52], regardless of the used load factor.

For linear probing, the throughput depends on the length of the sequential scan needed to handle collisions. We observe that the

[7]Note that PGM has high inference cost because of its multi-level structure.

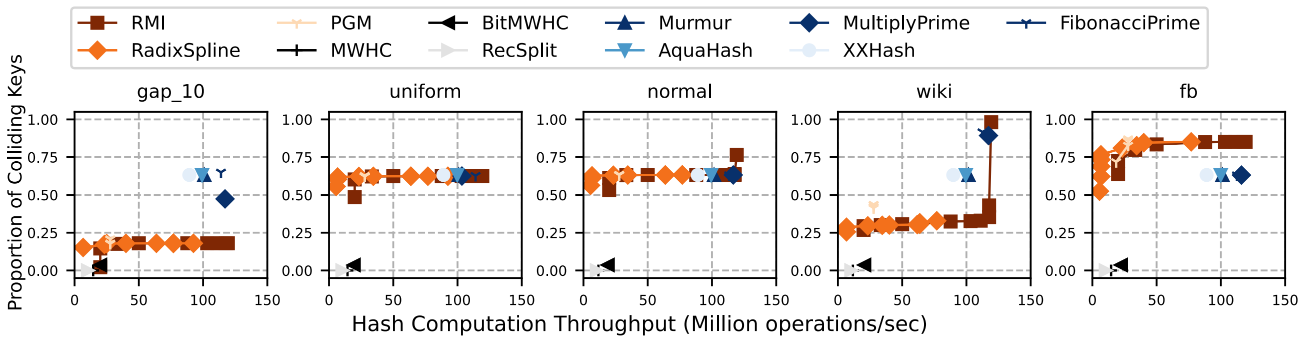

Figure 3: Computation throughput and collisions tradeoffs for various hash functions and using different datasets.

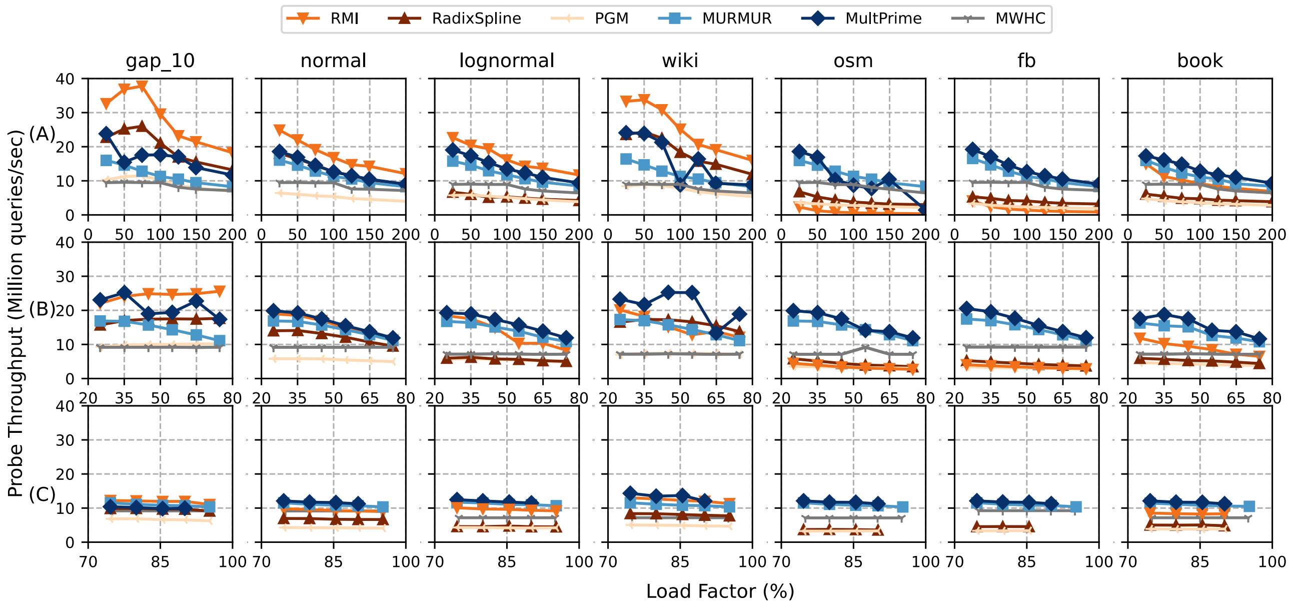

Figure 4: Probe throughput for combinations of 6 hash functions and 3 hashing schemes: (A) bucket chaining, (B) linear probing, and (C) cuckoo hashing. Results are shown for 7 different datasets, and various load factors for each hashing scheme.

throughputs achieved by using Murmur, MultiplyPrime, and MWHC have the same trend as in bucket chaining. In contrast, RMI and RadixSpline have the following two notable changes. First, their performance gain over traditional hashing in *gap_10* decreases or even vanishes (e.g., they yield 10% less throughput in *normal* and *lognormal*). Although the number of collisions using *LMH* functions is slightly smaller in these datasets (Section 7.2), the effect of this difference can be hidden by the sequential scan benefits (e.g., prefetching) of linear probing, and hence the overhead of RMI and RadixSpline hash computation becomes more significant. Second, RMI and RadixSpline result in worse throughput than traditional hashing in *wiki* (average 40% less throughput than MultiplyPrime). This was a bit surprising as *LMH* functions result in significantly fewer collisions than traditional hashing. However, we found that in a few parts of the *wiki* dataset RMI maps up to 100 keys to the same slot, creating clusters that result in long sequential scans during probing.

For cuckoo hashing, we observe that the throughputs achieved by any hash function are pretty much similar within the same dataset, regardless of the load factor used. This is expected as handling collisions in cuckoo hashing is typically performed in constant time (two cache misses at most). Even better, we employ a biased kicking strategy, in which most of the tuples are placed in their primary hash slots (i.e., one cache miss for most of the probes). This makes the hash function computation (model prediction in case of *LMH* functions) have a great impact on the probe latency in cuckoo hashing, and explains why the throughput using *LMH* functions is worse than using traditional hashing in all datasets, except in *gap_10* and *wiki* where RMI is almost similar to Murmur. Note that using RMI failed to construct the cuckoo hash table for *fb* and *osm* (similarly, RadixSpline and PGM failed in *fb*, *osm*, and *book* at load factors > 90%) because the resulting number of collisions is extremely high, and the required number of kicks to handle them exceeds the maximum threshold. For traditional hashing, we also noticed that Murmur

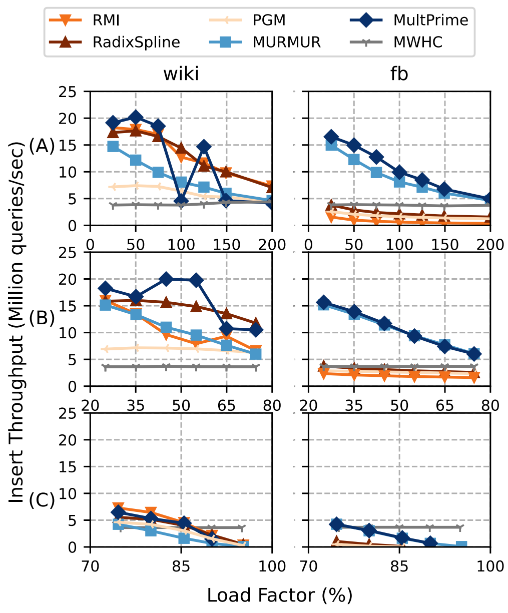

Figure 5: Insert throughput for the same hash functions and schemes used in Figure 4, yet for *wiki* and *fb* datasets only.

succeeded in constructing the hash tables in all datasets, while the construction failed using MultiplyPrime at load factor 95% because of high number of collisions. In general, cuckoo hashing significantly reduces the impact of collisions, regardless of the hash function used, and hence the performance improvement of *LMH* over traditional hashing becomes negligible.

Insert Throughput. Here, we use the same setup in the probe throughput experiment, while changing the query workload. To generate the insert workload, we first uniformly and randomly sample 101 million tuples from an input dataset. Then, we initialize the hash table by bulk-inserting 100 million tuples from this sample as in the probe throughput experiment, and use the remaining 1 million tuples as the query workload. Figure 5 shows the results of this experiment for two input datasets only, *wiki* and *fb* (the remaining datasets show similar performance trends).

In general, the relative ranking and throughput trends remain the same as in the probe throughput experiment (including the failure cases in cuckoo hashing). We also observe that, in *wiki*, the performance benefit that RMI offers over MultiplyPrime - when used with bucket chaining - in insertion is not as high as in probing (only an average of 10% throughput improvement in insertion compared to 30% in probing). Probing time mainly depends on the length of the chain to be traversed whereas insertion requires allocating and adding new buckets to the chain, and hence the collision reduction improves only a portion of the total insert time. Another interesting observation is that at load factor 95% using cuckoo hashing with MWHC is the best as the overhead of kicking operations becomes higher than the complex computation of MWHC.

Performance Counters. To deeply understand what happens on the hardware level, we investigate the following four performance counters: cycles, L1 cache misses, last-level cache (LLC) misses, and branch misses. Figure 6 shows the average values of these counters per tuple when using RMI, MultiplyPrime and MWHC in the probe

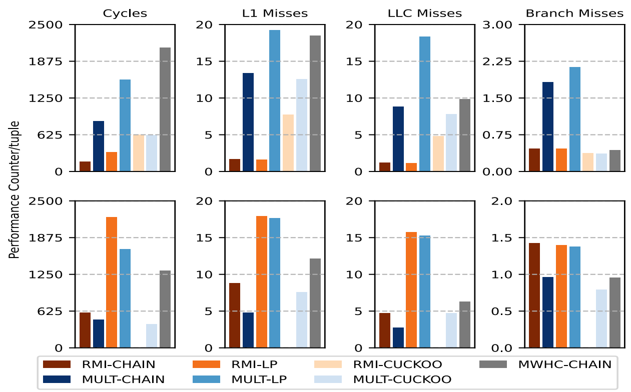

Figure 6: Performance counters per tuple for the probe experiment in Figure 4 using the *gap_10* (first row) and *fb* (second row) datasets at load factor 80%.

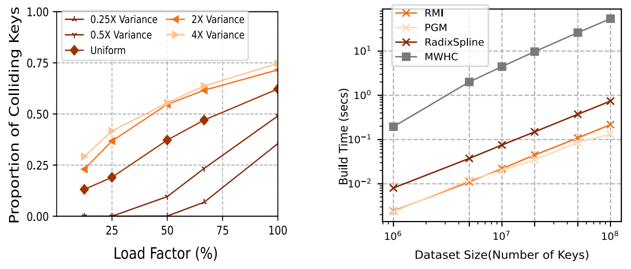

Figure 7: Effect of (1) gap distribution on *LMH* collisions (left), and (2) dataset size on building time (right).

throughput experiment (Figure 4) at load factor 80% and only for two datasets *gap_10* (first row) and *fb* (second row). For MWHC, we only show chained results as other schemes have similar performance.

For *gap_10*, the three RMI-based variants achieve the lowest performance counter values (e.g., one L1/LLC miss per tuple for RMI-CHAIN and RMI-LP) compared to other variants. This is because the number of submodels needed for any *LMH* function is only 10 (as shown in Table 1), which can totally fit in the cache. For *fb*, we found that scanning very large clusters, as in RMI-LP or MULT-LP, significantly increases both cache and branch misses, and in turn increases cycles (high cache and branch misses lead to an excessive increase in the amount of CPU stalls and wasted cycles, respectively). In contrast, RMI-CHAIN significantly reduces the effect of the high collisions produced by RMI in *fb* (RMI-CHAIN has at least 3X less LLC misses and cycles than RMI-LP). Even in *gap_10*, RMI-CHAIN still has at least 2X and 4X less cycles than RMI-LP and RMI-CUCKOO, respectively. This confirms our conclusion about the impact of hashing schemes on the probe throughput using *LMH* functions. Another interesting observation in *fb* is that MULT-CHAIN and MULT-CUCKOO have close values in all counters, yet MULT-CUCKOO is a bit better in cycles and branch misses. This shows that bucket chaining can provide a competitive performance at challenging datasets and high load factors.

7.4 More Performance Analysis

In this section, we study more parameters related to *LMH* functions and their performance in hash tables.

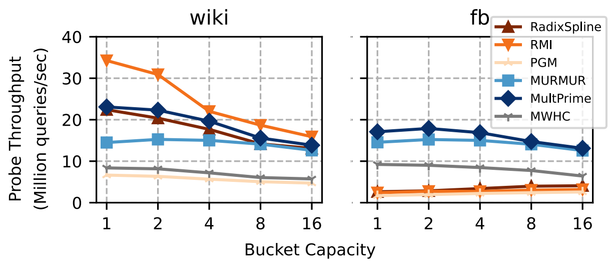

Figure 8: Effect of increasing the bucket capacity on the probe throughput of a chained hash table at load factor of 50%.

Gap Distribution. In this experiment, we vary the gap distribution to display that gaps concentrated around the mean have lower collisions. Assuming that the variance of the gap distribution of *uniform* keys is X, we generate 4 different variations of the *uniform* dataset, such that the gap distribution variances of their keys are $2X$, $4X$, $0.5X$ and $0.25X$ (i.e., scaled variances) [8]. Then, we insert the keys of each dataset variation in a hash table using RMI, and calculate the proportion of colliding keys. The left part of Figure 7 shows the proportion of colliding keys with varying load factors. As expected, the amount of collisions can be decreased by decreasing either decreasing the gap variance or the load factor. Lower gap variance cause the gap distribution to concentrate around the mean value resulting in lower collisions.

Build Time. Unlike traditional hash functions, *LMH* and perfect hash functions require a building stage. In the right part of Figure 7, we show the building time for *LMH* and MWHC functions using the *uniform* dataset, varying the number of keys between 10^6 and 10^8. We see that the building time of MWHC is consistently 2.5 and 2 orders of magnitude slower than the building times of RMI (or PGM) and RadixSpline, respectively. Although MWHC has an *expected* $O(n)$ construction time [52], its hypergraph building process requires an excessive amount of random memory accesses, and hence cache misses (check Section 4). In contrast, building *LMH* functions requires only sorting the data once and doing multiple sequential passes over it, which is a cache-friendly process.

Bucket Capacity. In this experiment, we study how increasing the bucket capacity (i.e., number of tuples in the bucket) affects the probe throughput. For each dataset, we build different hash tables with a load factor of 50%, and are bulk-loaded with 100 million tuples. Note, since we fix the load factor, increasing the bucket capacity by a factor X reduces the number of buckets by a factor $\frac{1}{X}$. We use the same inserted tuples as a probe workload, after randomly shuffling them, and measure the throughput as in Figure 8.

In bucket chaining, increasing the bucket capacity reduces the length of needed chains (i.e., extra buckets) to handle collisions, as any colliding key now has a high probability to be in the main hash table bucket. However, this increases the probe time as well because finding a key in the bucket requires larger scan overhead as the bucket becomes larger. In *wiki*, *LMH* functions already produce a low number of collisions, and hence increasing the bucket capacity will not benefit chaining, yet causes probes to scan unnecessary keys, and hence the throughput significantly decreases (this is also true

[8]Each dataset variation is generated by scaling the gaps between the *uniform* keys with the corresponding factor (e.g., the "$2X$ Variance" dataset scales the gaps between *uniform* keys by a factor of 2).

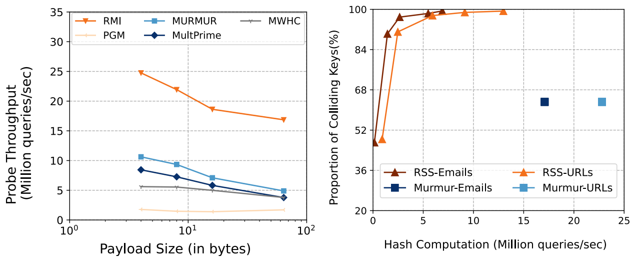

Figure 9: (1) Effect of payload size on the probe throughput (left), and (2) Computation throughput and collisions tradeoff when using string keys (right).

for MWHC as it has no collisions by definition). We can see that RMI and RadixSpline are affected more than PGM because their hash computation is lighter, and hence the effect of collision handling, with any extra overhead, becomes more obvious in the total probe time. In *fb*, *LMH* produces a lot of collisions that result in longer chains. In this case, increasing the bucket capacity improves the probe throughput a bit. In the case of Murmur and MultiplyPrime, they significantly suffer from the extra scan overhead within the bucket only beyond size of 4.

Payload Size. Here, we study the effect of increasing the payload size on the probe throughput of hash tables built with different hash functions. In this experiment, we use the *wiki* dataset and a chaining scheme with a 100% load factor. For each hash function, the hash tables are built and probed as described in the bucket capacity experiment, yet, with tuples of different payload sizes: 4, 8, 16 and 64 bytes. The left part of Figure 9 shows the probe throughput (x-axis has a logscale). As expected, increasing the payload size significantly reduces the probe throughput of all functions, except MWHC and PGM, in which the overhead of cache misses (coming from accessing payloads) does not affect their already high computation time until the payloads become very large (e.g., 64 bytes).

String Keys. Workloads with string keys are common in the real-world (e.g., [5, 16]). Unfortunately, learned models and indexes have no efficient support for string keys. The most relevant work in this area is RadixStringSpline (RSS) [77], which constructs a tree of radix splines, each indexing a fixed number of bytes in the string. Here, we investigate the robustness of RSS against Murmur, which is known for its efficiency in hashing strings. We repeat the computation throughput-collisions experiment (Section 7.2), while using two string datasets: (1) *Emails*, which is a real-world email dataset used in [12], and (2) *URLs*, which is a dataset of Wikipedia URL tails used in [77]. The right part of Figure 9 shows the results of this experiment. We can see that RSS, at its best configurations, can actually achieve 28% lower collisions than Murmur but with extremely slow computation throughputs. This slowness is because strings typically have both long shared prefixes and relatively low discriminative content per byte, which require a very large number of submodels (i.e., tree nodes) in the RSS to capture the strings distribution.

7.5 Range Queries Performance

Hash tables support fast point queries only, and do not support range queries. On the other hand, index structures like B-Tree, ART [47], and RMI [43] support both point and range queries. However, the performance of index structures in point queries is not as efficient as

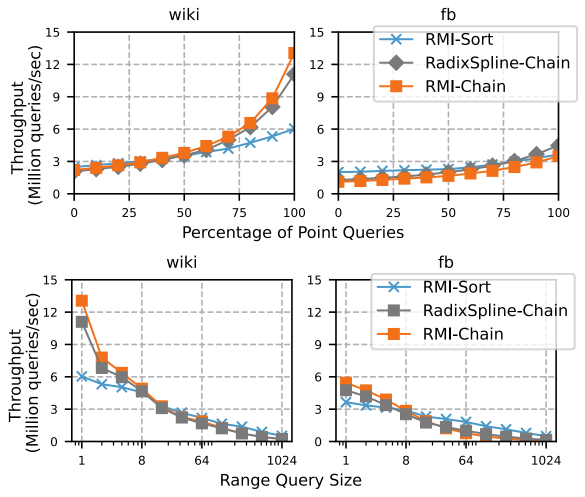

Figure 10: Effect of both point queries percentage (first row), and range query size (second row) on the queries throughput.

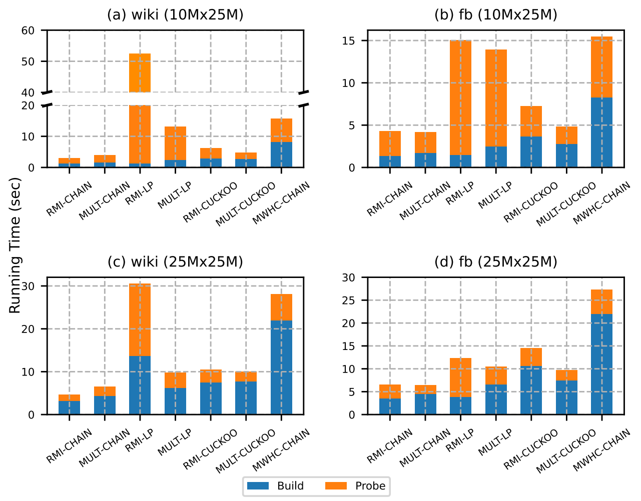

Figure 11: Runtime breakdown for the different implementations of non-partitioned hash join using various hash tables.

hash tables. In case of having a mixed workload of point and range queries, where range queries represent only a small proportion, one cannot use a hash table and is forced to use an index to be able to answer the range queries. This results in a huge performance degradation for the majority of the point queries. Fortunately, we can use "monotonic" *LMH* functions, such as RMI and RadixSpline, along with bucket chaining to build a hash table that supports range queries in addition to its natural support for fast point queries. In this case, a range query can be processed by scanning the buckets between the locations corresponding to the query lower and upper bound keys. This is possible as monotonic *LMH* functions are order-preserving, and hence the target keys are bound to be within the bucket locations.

Point Queries Percentage. In this experiment, we study the throughput of a mixed workload of point and range queries (the percentage of point queries is variable) using (1) RMI-CHAIN and RadixSpline-CHAIN hash tables (bucket size of 8 and load factor of 50%), which are our proposed solutions, and (2) a sorted array of the input data with a typical RMI on top of it (we refer to it here as RMI-SORT). We use *wiki* and *fb*, where we sample 100 million tuples from each one of them as input data. We generate a mixed query workload by first randomly sampling X% of the input data to be used as point queries, and then for the rest of the workload (i.e., 100-X%) we generate random range queries that retrieve about 25-50 tuples. The upper part of Figure 10 shows the results for this experiment, where we vary X between 0 and 100. As expected, in both *wiki* and *fb*, RMI-CHAIN and RadixSpline-CHAIN have faster throughput than RMI-SORT when the workload has a majority of point queries, and vice versa. This is explainable as, for a point query, they just need to scan the bucket pointed out by the model, whereas RMI-SORT needs a local search to find the relevant key. For a range query, RMI-CHAIN and RadixSpline-CHAIN scan the buckets that fall within the range query and also the additional chains associated with them. This leads to excessive random memory accesses, and hence a decrease in the throughput. In contrast, RMI-SORT is more suitable for range queries as it only needs to sequentially scan the relevant keys in the sorted range.

Range Query Size. Here, we reuse the setup of the previous experiment, while focusing only on 100% range queries workload. We vary the range query size from 1 to 1024. The below part of Figure 10 shows the results for this experiment (x-axis has a logscale). With increasing the range size, RMI-CHAIN and RadixSpline-CHAIN become slower than RMI-SORT as they need to scan additional chained buckets.

7.6 Hash-based Join Performance

In this experiment, we are interested in understanding the performance of non-partitioned hash join (NPJ) over two input relations [45, 80], when implementing it using different combinations of hashing functions and schemes. Note that we do not investigate the hashing effect on partitioned hash join [73, 80] as it employs small cache-fit hash tables. In this situation, using any traditional hash function to build such small tables will be the best choice. In contrast, NPJ builds a large global hash table for the smaller input relation and the probability of having performance degradation, due to large number of collisions, is high. Therefore, employing an efficient hash function is crucial to improve the join performance. We use *wiki* and *fb* datasets, where we uniformly and randomly sample two variations from each dataset with 10M and 25M tuples. These variations will be used to perform the NPJ on, using the best function in each hashing category. Figure 11 shows the running time of the NPJ build and probe phases.

Interestingly, we can observe that RMI-CHAIN and MULT-CHAIN have the best join performance in both *wiki* and *fb*. Specifically, RMI-CHAIN has 28% less total runtime (build and probe) than MULT-CHAIN in *wiki* (e.g., in the 25Mx25M variant, the build times of RMI-CHAIN and MULT-CHAIN are 3.2 and 4.3 sec, while their probe times are 1.488 and 2.193 sec, respectively), and they both have the same total runtime in *fb*. Looking at the build phase, we can see that RMI-CHAIN and RMI-LP build the hash table more efficiently than other solutions in most of the cases. This is mainly because RMI sorts the data to build its submodels, and then uses them to insert each tuple from the sorted data into the hash table. Although sorting the data is a bit expensive, it helps the model-based insertion to happen in a cache-friendly manner, and the overall overhead, including both sorting and model-based insertion, is still significantly less than randomly inserting tuples (i.e., more cache misses) using MultiplyPrime and Murmur. This observation was confirmed in a previous study before [44]. Due to the efficiency of RMI in building the hash table, the total time of NPJ using RMI-CHAIN becomes more competitive with MULT-CHAIN in a challenging dataset like *fb* because the performance gain in building compensates for the performance degradation in probing, and the total running time becomes very close.

8 RELATED WORK

Traditional Hashing. Traditional hash functions can be categorized as either non-cryptographic [29] or cryptographic [1]. *Non-cryptographic* hash functions [2, 18–20, 42, 67, 69, 71], which we mainly focus on in our study, are mostly used in building data structures and algorithms due to their good balance between computation time and collision rates. More recent work has focused on optimizing the performance of non-cryptographic hashing on modern hardware by either proposing new hash functions [81] (e.g., CLHash [48], and tabulation hashing [65]) or customizing the existing ones to utilize the underlying hardware (e.g., GPU [49] and SIMD vectorization [9, 34]). *Cryptographic* hash functions have the property of being computationally hard to invert. These functions can still be used in building data structures, yet their performance can be much slower than non-cryptographic ones [18]. Examples of cryptographic hash functions include MD5 [68], SHA1 [27] and SipHash [7].

Perfect Hashing. Perfect hashing has been widely studied; see, e.g., the survey in [52]. Perfect hashing solutions can be divided into two categories: *static* and *dynamic*. When inserting new tuples to the hash table, the static solutions (e.g., [14, 28, 52, 62]) reconstruct the whole table from scratch, while the dynamic solutions (e.g., [21, 82]) reconstruct the table parts that are related to the update only. Another interesting line of work is improving the perfect hashing computation using modern hardware, such as GPU (refer to a survey in [49]).

Learned Models for Indexing and Hashing. During the last few years, the idea of using CDF-based learned models to replace traditional indexes has been investigated extensively including single-dimension (e.g., [30, 40, 43]), multi-dimensional (e.g., [24, 58]), updatable (e.g., [25]), and spatial (e.g., [50, 63, 66]) indexes. Interestingly, the authors of [43] also discussed the idea of using learned models as order-preserving hash functions. A recent study [71] initially investigated whether learned models are better than traditional hash functions in performing hash table lookups or not. In contrast, our proposed study is more comprehensive as it spans additional hash function types, hashing schemes, workload types, and hash-based operations. Another recent interesting work [36] employs an entropy-learned approach to reduce the hashing overhead by choosing how much and which parts of the input data we need to hash.

Hashing Experimental Studies. Previous experimental studies for the performance of different hash functions and schemes have been provided. SMHasher [81] is a widely-known test-suite for evaluating non-cryptographic hash functions. [79] provided both theoretical and experimental analysis for cryptographic hash functions. [2] did a detailed experimental comparison between the performance of two hashing schemes (cuckoo hashing and quadratic probing) and two radix tree variations. [74] micro-benchmarked the performance of SIMD-aware variations of different hashing schemes. [67] is another recent comprehensive experimental study for the different combinations of hash functions and schemes. However, it only focused on non-cryptographic traditional hash functions and hash table operations. For learned models, they have been extensively benchmarked in [53] for indexing only, and not for hashing. In this paper, we try to fill this gap.

9 LESSONS LEARNED AND FUTURE WORK

Gaps distribution matters for *LMH* collisions. Assuming the input keys are sorted, collisions of *LMH* functions (that employ linear submodels) depend on the distribution of gaps between consecutive sorted keys. The more evenly spaced these gaps are, the fewer collisions *LMH* functions have; in fact there can be fewer collisions than traditional hashing. When data is from typical distributions (e.g., normal), we observe that *LMH* functions have a similar (or even higher) number of collisions compared to traditional hashing.

Number of submodels matters for *LMH* efficiency. Building an *LMH* function with very few submodels (fewer than a certain threshold) reduces its accuracy, as it will not capture the input distribution effectively. On the other hand, significantly increasing the number of submodels decreases the computation throughput. Accordingly, the number of submodels should be carefully tuned. For datasets with evenly spaced gaps, tuned *LMH* functions can achieve the best tradeoff between computation throughput and collisions, compared to traditional and perfect hash functions, as shown in Section 7.2. Generally speaking, RMI is the best in achieving this tradeoff, while PGM is the worst. For RadixSpline, it depends on the dataset skewness. The more skewed the dataset is, the worse the RadixSpline performance.

Throughputs of building/probing hash tables using *LMH* functions vary across different hashing schemes. Collision reduction due to *LMH* translates to improved hash table probe and insert throughputs. However, such improvements become more evident with bucket chaining, and diminish with cuckoo hashing, as shown in Section 7.3. We also found that RMI and MultiplyPrime are the best *LMH* and traditional hash functions, respectively, to use along with bucket chaining in all load factors ranging from 20% to 200%.

For string keys, traditional hashing is better than *LMH*. Efficient support for strings in learned models is still an open research question. RadixStringSpline [77], which is a preliminary attempt to support strings with radix splines, shows a significantly less hash computation throughput than traditional hashing due to the difficulty of modeling strings that typically have long shared prefixes and relatively low discriminative content per byte.

Efficient support of *LMH* for mixed workloads of point and range queries as well as non-partitioned hash join (NPJ). Any monotonic (i.e., order-preserving) *LMH* function, along with bucket chaining, can be used to build one hash table for efficiently answering both point and range queries at the same time (i.e., mixed workloads). In fact, using *LMH* functions along with chaining is also more efficient than other traditional options in building and probing the shared hash table in NPJ. Among the different *LMH*-based variants, RMI-CHAIN shows the most efficient and robust performance.

Future Directions. The multi-threaded implementations and evaluations of *LMH*, perfect hashing, and traditional hash tables remain worthy of further exploration. Designing a loss function for *LMH* that minimizes collisions while also suitably providing a tradeoff with hash computation time would be an interesting direction. Also, our study primarily focused on piece-wise linear models, and more complex models like decision trees and neural networks may lead to different and further interesting tradeoffs.

REFERENCES

[1] Mohammad Alahmad and Imad Fakhri Taha Alshaikhli. Broad View of Cryptographic Hash Functions. *International Journal of Computer Science Issues*, 2013.

[2] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. A Comparison of Adaptive Radix Trees and Hash Tables. In *ICDE*, pages 1227–1238, 2015.

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

[4] Austin Appleby. MurmurHash. https://sites.google.com/site/murmurhash/, 2011.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[6] Audouin Audouin and Brongniart Brongniart. Annales des sciences naturelles-vol. 7 (series-2). In *Annales des Sciences Naturelles*, volume 7, pages 42–110. Crochard, 1837.

[7] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT*, 2012.

[8] 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.

[9] Tobias Behrens, Viktor Rosenfeld, Jonas Traub, Sebastian Breß, and Volker Markl. Efficient SIMD Vectorization for Hashing in OpenCL. In *EDBT*, 2018.

[10] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *Journal of Experimental Algorithmics (JEA)*, 16:3–1, 2008.

[11] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Linear Probing Revisited: Tombstones Mark the Death of Primary Clustering. In *IEEE Symposium on Foundations of Computer Science*, 2021.

[12] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*, 2018.

[13] C++ Team Blog. Linker Throughput Improvement in Visual Studio 2019. https://devblogs.microsoft.com/cppblog/linker-throughput-improvement-in-visual-studio-2019/, 2019.

[14] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and Space-Efficient Minimal Perfect Hash Functions. In *Proceedings of the International Conference on Algorithms and Data Structures*, 2007.

[15] Fabiano C Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 653–662, 2007.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2020.

[17] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, CAN, 1986.

[18] Yann Collet. xxHash repository. https://cyan4973.github.io/xxHash/.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[20] 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.

[21] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[22] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 615–627. Springer, 2011.

[23] Martin Dietzfelbinger and Christoph Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. In *Theortical Computer Science*, 2007.

[24] Jialin Ding et al. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. In *Proc. VLDB Endow.*, 2020.

[25] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*, 2020.

[26] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*, 2017.

[27] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, IETF, 9 2001.

[28] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185. SIAM, 2020.

[29] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the Most Common Non-Cryptographic Hash functions. *Softw. Pract. Exper.*, 44(6):681–698, 2014.

[30] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020.

[31] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science*, 2003.

[32] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[33] Shay Gueron. 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.

[34] Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero Durand, and Gunter Saake. SIMD Vectorized Hashing for Grouped Aggregation. In *Advances in Databases and Information Systems*, 2018.

[35] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 317–326. Springer, 2001.

[36] Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-Learned Hashing: 10x Faster Hashing with Controllable Uniformity. In *SIGMOD*, 2022.

[37] 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.

[38] 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.

[39] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*, 2019.

[40] 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.

[41] Oliver Knill. Probability and stochastic processes with applications. *Havard Web-Based*, page 5, 1994.

[42] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

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

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

[45] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-Aware Hash Joins. In *In-Memory Data Management and Analysis, IMDM*, 2015.

[46] Sylvain Lefebvre and Hugues Hoppe. Perfect Spatial Hashing. *ACM Transactions on Graphics.*, 25(3):579–588, 2006.

[47] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, 2013.

[48] Daniel Lemire and Owen Kaser. Faster 64-bit Universal Hashing Using Carry-less Multiplications. *Journal of Cryptographic Engineering*, 6:171–185, 2015.

[49] Brenton Lessley and Hank Childs. Data-Parallel Hashing Techniques for GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 31(1), 2020.

[50] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*, 2020.

[51] PerfEvent Library. PerfEvent Library. https://github.com/viktorleis/perfevent, 2019.

[52] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[53] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. In *Proc. VLDB Endow.*, 2020.

[54] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. In *SIGMOD*, 2021.

[55] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.* Cambridge university press, 2017.

[56] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. ntHash: Recursive Nucleotide Hashing. *Bioinformatics*, 32(22):3492–3494, 2016.

[57] Michael Molloy. Cores in random hypergraphs and boolean formulas. *Random Structures & Algorithms*, 27(1):124–135, 2005.

[58] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning Multi-Dimensional Indexes. In *SIGMOD*, 2020.

[59] Thomas Neumann and Sebastian Michel. Smooth interpolating histograms with error guarantees. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases*, BNCOD '08, pages 126–138, 2008.

[60] Anna Pagh and Rasmus Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.

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

[62] Shekhar Palit and Kevin A. Wortman. Perfect Tabular Hashing in Pseudolinear Time. In *IEEE Annual Computing and Communication Workshop and Conference (CCWC)*, 2021.

[63] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. The Case for Learned Spatial Indexes. In *Proceedings of the AIDB Workshop @VLDB*, 2020.

[64] Orestis Polychroniou and Kenneth A. Ross. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *SIGMOD*, 2014.

[65] Mihai Pundefinedtraşcu and Mikkel Thorup. The Power of Simple Tabulation Hashing. *Journal of the ACM*, 59(3), 2012.

[66] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively Learning Spatial Indices. In *VLDB*, 2020.

[67] 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.

[68] Ronald L. Rivest. The MD5 Message-Digest Algorithm. *RFC*, 1321:1–21, 1992.

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

[70] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *SIGMOD*, page 1228–1242, 2022.

[71] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When Are Learned Models Better Than Hash Functions? In *Proceedings of the AIDB Workshop @VLDB*, 2021.

[72] David Salomon. Data compression. In *Handbook of massive data sets*, pages 245–309. Springer, 2002.

[73] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*, 2016.

[74] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K. DK Panda. SimdHT-Bench: Characterizing SIMD-Aware Hash Table Designs on Emerging CPU Architectures. In *IEEE International Symposium on Workload Characterization, IISWC*, 2019.

[75] Malte Skarupke. Fibonacci Hashing: The Optimization that the World Forgot (or: a Better Alternative to Integer Modulo). https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/.

[76] Vera T Sós. On the theory of diophantine approximations. i 1 (on a problem of a. ostrowski). *Acta Mathematica Hungarica*, 8(3-4):461–472, 1957.

[77] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. Bounding the Last Mile: Efficient Learned String Indexing. In *Proceedings of the AIDB Workshop @VLDB*, 2021.

[78] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday Paradox for Multi-Collisions. In *Proceedings of the International Conference on Information Security and Cryptology*, 2006.

[79] Jacek Tchórzewski and Agnieszka Jakóbik. Theoretical and Experimental Analysis of Cryptographic Hash Functions. *Journal of Telecommunications and Information Technology*, 2019.

[80] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*, 2013.

[81] Reini Urban. Smhasher. https://github.com/rurban/smhasher.

[82] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. MapEmbed: Perfect Hashing with High Load Factor and Fast Update. In *SIGKDD*, 2021.

[83] S. Świerczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Mathematicae*, 46(2):187–189, 1958.