Entropy-Learned Hashing
Constant Time Hashing with Controllable Uniformity

Brian Hentschel
Harvard University

Utku Sirin
Harvard University

Stratos Idreos
Harvard University

ABSTRACT

Hashing is a widely used technique for creating uniformly random numbers from arbitrary input data. It is a core component in relational data systems, key-value stores, compilers, networks and many more areas used for a wide range of operations including indexing, partitioning, filters, and sketches. Due to both the computational and data heavy nature of hashing in such operations, numerous recent studies observe that hashing emerges as a core bottleneck in modern systems. For example, a typical complex database query (TPC-H) could spend 50% of its total cost in hash tables, while Google spends at least 2% of its total computational cost across all systems on C++ hash tables, resulting in a massive yearly footprint coming from just a single operation.

In this paper we propose a new method, called Entropy-Learned Hashing, which reduces the computational cost of hashing by up to an order of magnitude. The key question we ask is "how much randomness is needed?": We look at hashing from a pseudorandom point of view, wherein hashing is viewed as extracting randomness from a data source to create random outputs and we show that state-of-the-art hash functions do too much work. Entropy-Learned Hashing 1) models and estimates the randomness (entropy) of the input data, and then 2) creates data-specific hash functions that use only the parts of the data that are needed to differentiate the outputs. Thus the resulting hash functions can minimize the amount of computation needed while we prove that they act similarly to traditional hash functions in terms of the uniformity of their outputs. We test Entropy-Learned Hashing across diverse and core hashing operations such as hash tables, Bloom filters, and partitioning and we observe an increase in throughput in the order of 3.7X, 4.0X, and 14X respectively compared to the best in-class hash functions and implementations used at scale by Google and Meta.

KEYWORDS

hashing, hash tables, Bloom filters, point indexing

ACM Reference Format:
Brian Hentschel, Utku Sirin, and Stratos Idreos. 2022. Entropy-Learned Hashing Constant Time Hashing with Controllable Uniformity. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517894

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517894

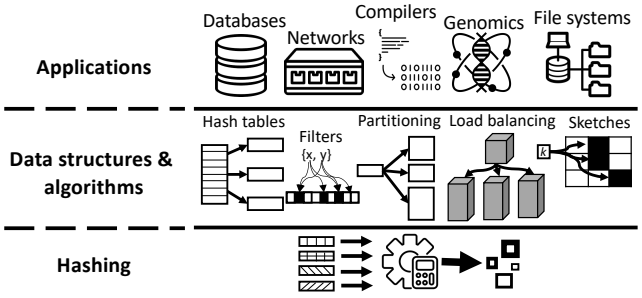

Figure 1: Hashing is a core element for numerous fundamental components across diverse classes of systems.

1 DATASET SPECIFIC HASHING

Hashing is Central to Computer Systems. Hashing is one of the core concepts in computer science; data structures and algorithms which use hashing exist in nearly every computer program. It's most ubiquitous use case, hash tables, is the standard way to access individual data items. They are used both for fast access to hot data in L1 cache across general purpose programs as well as for accessing colder data that lies outside of cache either in memory or on disk. For example, in relational database systems hash tables are used for joins and group by operations. Beyond hash tables, hashing is used in numerous other core parts of computer science such as filters [11], data partitioning [55], load balancing [42], and sketches [15, 25]. As a result of their many and important use cases, hashing is not only central within relational databases [58, 59] but acts as a core component of systems across compilers [3], file systems [63, 69], gaming [28], genomics [43], and more. This effect is depicted visually in Figure 1 where hashing is shown as the core design element used to build numerous fundamental operations, data structures, and algorithms (hash tables, filters, partitioning, etc.) which in turn are are core components of diverse systems.

Hashing: Expensive at Scale. Because hashing is so ubiquitous, hash functions and their uses are a substantial portion of overall system cost. Google states that 2% of its total CPU usage and 5% of its total RAM at the company is spent on just one hash-based data structure, hash tables, in just one of the languages used, C++ [36]. Including other languages and other hash-based operations, the total CPU and memory usage spent on hashing overall is surely much higher. Meta makes similar statements, with developers stating that hash tables are such "a ubiquitous tool in computer science that even incremental improvements have large impact" [16]. Moving from large cloud infrastructure to particular applications, inside databases hash-based joins and aggregations are amongst the most expensive and used operators; as a concrete example they account for over 50% of total time on 17 of the 22 queries on the TPC-H benchmark for the Hyrise DBMS [23, 62]. A second example can be seen in compilers, where using hash tables in linking is a substantial

part of program compilation costs in Visual Studio [3]. Moving beyond hash tables, filters are a key computational bottleneck in LSM trees [21, 71], while similarly sketches act as a key computational bottleneck in network switches [40]

These observations across diverse industries, systems and data structures spell out an important fact: *despite numerous algorithmic and engineering advances, hashing use cases are still expensive because of the frequency and scale at which they are used.*

Randomness vs. Performance. To start drilling in at both the source of the problem and our solution we will next discuss the core mechanisms and trade-offs in hashing. A core component of all hash-based data structures and algorithms is the hash function itself, with hash functions having two primary goals. The first is to create uniformly random outputs for any number of input items. That is, the output should be jointly uniform as well as marginally uniform. The second is computational efficiency. While ideally both goals would be optimally achievable, these two goals are practically at odds with each other. Thus a central question for all hash-based operations is how much randomness is needed from the hash function for the operation at hand.

Guarantees Without Assumptions on the Data. To get performance guarantees on hash data structure performance without assumptions on the data, all randomness needs to come from the hash function. The main way to define this property is by bounding the likelihood of collision for arbitrary input items. In universal hashing [17], one guarantees that for any two items $G \neq \sim$ and family of functions $: f0 \bullet 1g^{=} \; ! \; f0 \bullet 1g^{<}$, the probability when choosing a random from of $^1G^o = {}^1\sim^o$ is $\frac{1}{<}$. However, this is not enough randomness for many data structures [48, 50], and so an expanded idea of hash randomness is k-independence, which is that for any set of $:$ inputs $G_1 \bullet """ \bullet G_:$, and $:$ outputs $\sim_1 \bullet """ \bullet \sim_:$, the probability of $\%^1\backslash_8 \; ^1G_8{}^o = \sim_8{}^o = < \; :$ [68]. Given this model, it becomes possible to provide guarantees about data structures and algorithms, with larger amounts of independence being more computationally expensive but providing better performance guarantees [48, 50, 68].

Hashing in Practice. In practice, systems designers avoid expensive k-independent hash functions and instead opt for hash functions which lack formal robustness guarantees but are faster to compute [7]. For instance, RocksDB uses xxHash [19], Google heavily uses CityHash, Wyhash, and FarmHash [53, 54, 66], and C++ compilers such as g++ often choose MurmurHash [1, 6]. This is because the computational performance of hashing is too important: systems designers are willing to give up concrete performance guarantees in exchange for faster hashing.

Another reason systems designers choose hash functions without formal guarantees is that empirically, their outputs appear as random as if they were from perfectly random hash functions [49, 56, 57]. One explanation for this phenomena is pseudorandomness. The main idea is the following: most hash functions perform well on most input data, and it takes careful manipulation of the input data to craft scenarios where commonly used hash functions fail. In other words, if we give up guaranteed performance on *all datasets* and instead assume *data itself is random enough*, then hash functions with weaker guarantees in terms of independence can be shown to perform in expectation nearly identically to those that are fully random [18, 41].

Problem Definition. Having given the core concepts in state-of-the-art hashing, we can now restate the problem more concretely. Modern systems across diverse areas and industries utilize fast hash functions but without any guarantees. However, these fast hash functions are still not fast enough: they are still slow in that they occupy a large portion of total cost in all those systems. In this paper, we ask the following question:

"Is it possible to improve on the speed of the best modern hash functions such that this brings significant end-to-end impact across diverse widely used hash-based operations, while at the same time maintaining and controlling their uniformity properties"?

The Solution: A Dataset-specific view of Hashing. Our core intuition is to utilize the inherent randomness in the data in a controlled way. That is, if we know how random the input data is, we can use this observed randomness to create faster hash functions by doing just enough computation and data movement to create a sufficiently random output. Our key insight is that hash functions in state-of-the-art solutions are "fixed" in that they always do the same work regardless of the input. As such they end up doing more work than needed if data sources are already random enough. Our goal is to utilize such "surplus randomness" in the data to minimize cost by adapting the hash function to the data.

Our resulting solution, *Entropy-Learned Hashing*, designs the hash function for a data source in two steps. In the first step, it uses samples of past data items and queries to estimate the amount of randomness in input keys at sets of byte locations. The second step then uses this learned randomness to choose subsets of bytes from input keys to hash. These subsets are chosen to have just enough randomness for the task at hand, creating faster hash functions while preserving the (approximate) uniformity of the hash function's output. For instance, for a dataset with input keys of length 120 bytes, if some consistent subset of bytes (such as bytes 3,7,9,12, and 15) is sufficiently random, Entropy-Learned Hashing computes a hash function using only these bytes and requires approximately only 1•24th the amount of computation.

Constant-Time Hashing. As a result of this view and its subsequent analysis, Entropy-Learned Hashing changes hashing from an operation whose runtime is linear in the size of input keys to one which is a constant-time operation with computational complexity independent of key size. Thus when compared to traditional hashing, it provides theoretical improvements which are unbounded as key sizes grow.

Contributions. The rest of the paper builds out the idea of Entropy-Learned Hashing, showing analytically and experimentally its improvements over traditional hashing. Specifically, we view our contributions as:

Entropy-Learned Hashing Formalization: We introduce a new way to design hash functions that uses the entropy inside the data source to reduce the computation required by hash functions.
Optimization: We show how to choose which bytes to hash given a collection of past queries and data items to analyze.
Generalization: We show how the entropy of partial-key hashes generalizes to data items outside the given sample of data.
Concrete Trade-offs: We derive metric equations for three core hash use cases of Entropy-Learned Hashing: hash tables, Bloom filters, and data partitioning. This allows users to trade-off speed

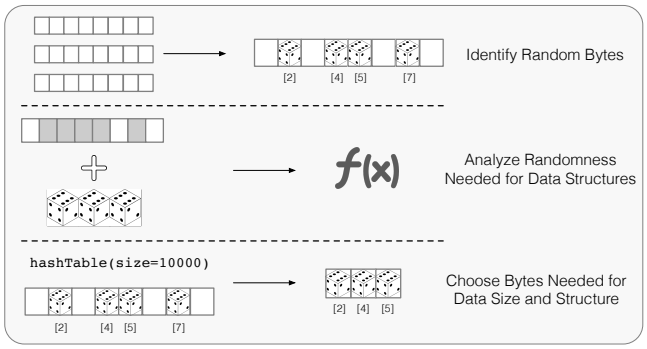

Figure 2: The core steps in Entropy-Learned Hashing.

in hash computation for small changes in other metrics of interest such as the number of comparisons, FPR, and partition variance.

Experiments: Comparing against state-of-the-art designs and implementations (e.g., Google's and Meta's hash tables), we show that Entropy-Learned Hashing provides faster overall throughput than traditional hashing. While this improvement is unbounded with key size, for common medium-sized key types such as URLs, we show this improvement is up to 3.7 for hash tables, 4.0 for Bloom filters, and up to 14 for data partitioning.

The paper is curated to be self-contained with the most critical material and we also accompany it with an online appendix with detailed proofs and numerous additional experiments [29].

2 OVERVIEW & MODELING

We now move on with a detailed description of Entropy-Learned Hashing which will span the next three sections. In this section, we start with a more detailed overview as well as laying out the basics for notation and modeling which we use throughout the paper.

Overview. The goal of Entropy-Learned Hashing is to learn how much randomness is needed and to produce a hash function which does just enough work by controlling the input given to the hash function. To achieve this goal, Entropy-Learned Hashing looks for bytes which are highly random on input objects and passes just enough of these bytes to create a highly random output. Stated more formally, Entropy-Learned Hashing consists of creating a hash function ^{0} which is the composition of 1) a partial-key function $!$ which maps a key G to any subkey of G (including potentially the full key G), and 2) , a traditional hash function. Our focus is on designing $!$, and can be any of the many well-engineered hash functions for full-keys.

In order to create the partial-key function $!$, Entropy-Learned Hashing uses three steps as shown in Figure 2. First, it analyzes the data source G and identifies which bytes are highly random, and how much entropy can be expected from a choice of $!$ (Section 3). Second, it reasons about how $!$ affects data structure metrics (Section 4). Finally, it uses runtime information, such as the size of the desired Bloom filter or hash table or the number of partitions in partitioning to choose which bytes to use in $!$ (Section 5).

Notation. The notation for all variables used is given in Table 1. Capital letters refer to either random variables or sets whereas lower case variables refer to fixed quantities. The new notation is because keys entered into are no longer unique. The set of keys contained in a hash-based data structure is broken down into

Notation	Definition (filter, hash table, or load balancer)
$-\bullet G$	key stored in the filter or hash table
$\bullet$	hash function for filter or hash table
$.\bullet\sim$	query key in filter or hash table
$<$	size of filter (in bits), table (in slots), or # bins
$=$	number of keys in filter or table
	set of keys
$($	multi-set of partial keys. Equal to ^{1} $^{!}\bullet I^{o}$
$^{!}$	Set of all partial keys.
I	maps each key G 2 $^{!}$ to $^{!}\,^{1}G^{o}$. I_G is used as shorthand for $I\,^{1}G^{o}$ throughout.

Notation	Definition (hash table only)
U	fill of hash table: $\frac{=}{<}$
$\%^{0}$	number of comparisons to find non-existing key
$\%$	average # of comparisons to retrieve a key in the dataset

Table 1: Notation used throughout the paper.

the multi-set $($ = ^{1} $^{!}\bullet I^{o}$. Here, $^{!}$ is the set of all partial-keys (outputs of $!$ applied to keys in), and I maps each key in $^{!}$ to the cardinality of its pre-image in . For instance, if $!$ takes the first two characters of an input and = {dog, dot, cat, fang}, then $^{!}$ = {"do"•"ca"•"fa"}, $I\,^{1}$"ca"^{o} = 1, and $I\,^{1}$"do"^{o} = 2.

Hash Function Model. We assume that is ideally random, i.e. that for any distinct inputs $G_1\bullet^{""}\bullet G_=$, output range $\ll^{1}/_{4}$ = {1$\bullet^{""}\bullet<$}, and outputs $O_1\bullet^{""}\bullet O_=$ 2 $\ll^{1}/_{4}$, we have

$$P^{1}\ \ ^{1}G_1{}^{o} = O_1\bullet^{""}\bullet\ \ ^{1}G_={}^{o} = O_={}^{o} = \prod_{\ell=1}^{=} P^{1}\ \ ^{1}G_\ell{}^{o} = O_\ell{}^{o}\ \ = {}^{1}\frac{1}{<}{}^{o=}$$

We do not use k-independent hashing; as noted before and as shown again in our experiments, hash functions tend to perform empirically like their perfectly random counterparts. Moreover, most proofs using k-independent hashing give big-O guarantees but drop constant factors [41, 48, 50]. These constant factors are of significant importance for high performance hash functions.

Source Model. Conditioned on $!$ we assume that the partial-keys $!\,^{1}-^{o}$ are i.i.d. distributed because the main metrics for hash-based algorithms tend to be order-independent. For instance, whether keys are ordered $G_1\bullet^{""}\bullet G_=$ or in the reverse order $G_=\bullet^{""}\bullet G_1$, the slots filled in a linear probing hash table or the length of the linked lists in a separate chaining hash table are identical. Similar statements hold for the false positive rate of Bloom filters and the partitions produced by partitioning. Thus, even if the original source has a temporal nature that might be better modelled by a Markovian assumption, the marginal distribution over time is more important.

3 CREATING PARTIAL-KEY FUNCTIONS

The first step is to create the partial-key function $!$ which needs knowledge about the data we expect. In the case of fixed datasets, such as read-only indexes like those used in the levels of LSM-based key-value stores [47], this is the actual dataset. With updates, we need a sample of past data and queries.

Metric for Partial-Key Hash Functions. Partial-key functions have two metrics. The first is the number of bytes in their output, with fewer being better so that subsequent hash computation is faster. The second is the Rényi Entropy of order 2 of their output, also known as the collision entropy. For a given discrete random

variable X, its Rényi Entropy of order 2 is $H_2(X) = -\log \sum_{i=1}^{n} p_i^2$ where p_i is the probability that X takes on the ith symbol in an alphabet $A = \{a_1, \ldots, a_n\}$. It draws its name from the fact that if X_1, X_2 are drawn i.i.d. from the same distribution as X, then $H_2(X) = -\log P(X_1 = X_2)$. We use collision probability to refer to $P(X_1 = X_2) = P(X_1 = X_2)$ and mean Rényi Entropy of order 2 whenever we use the term entropy. For Entropy-Learned Hashing, Rényi Entropy tells us how likely collisions are to occur. The following lemma will be useful in our analysis:

Lemma 1. *Given n i.i.d. samples from a distribution X, the number of observed collisions over the number of 2-combinations is an unbiased estimator of the collision probability for X. That is, if n_i is the number of times a symbol a_i appears in the sample, then we have*

$$E\left[\sum_i \frac{\binom{n_i}{2}}{\binom{n}{2}}\right] = \frac{\binom{n}{2}}{\binom{n}{2}} P(X_1 = X_2)$$

where $n^{\underline{2}} = n(n-1)$ is the 2nd falling power. Equivalently,

$$E\left[\sum_i \frac{\binom{n_i}{2}}{\binom{n}{2}}\right] = \frac{\binom{n}{2}}{\binom{n}{2}} 2^{-H_2(X)}$$

Proof. There are $\binom{n}{2}$ possible 2-combinations in n samples, each of which can produce a collision. The probability of collision is $2^{-H_2(X)}$ and so the expected number of collisions is $\binom{n}{2} P(X_1 = X_2)$. □

Optimization: Selecting the Bytes to Hash. The goal is to optimize our two metrics on our optimization set, which is either the fixed dataset or a training set of a sample of prior data items. Since our two metrics are at odds, the goal is to find an optimal Pareto frontier establishing for each $l = 1, 2, 3, \ldots$, what set of l bytes from our full-key input produces the most entropy.

Insight into this problem, as well as potential solutions, can be found by analyzing the similar problem for maximizing Shannon entropy (equivalently, Rényi entropy of order 1). In particular, for Shannon entropy selecting the best subset of size l of random variables from amongst n random variables is known to be NP-hard [33], suggesting that an optimal solution for Rényi entropy is likely computationally difficult. However, the greedy algorithm, described in detail below, is known to provide a $1 - \frac{1}{e}$ approximation to the best possible solution for Shannon Entropy because Shannon Entropy is submodular [44]. Additionally, real-life applications of the greedy algorithm tend to get solutions which are close to the optimal solution [9].

Inspired by this success and by the connectedness of Rényi and Shannon entropy, we use the greedy algorithm to optimize Rényi entropy on our training set. In particular, we start by using a dummy hash which reads zero bytes of the data items. Then, we continually add new bytes to the partial key function L in a way that decreases the number of collisions the most on the training data. After each new chunk of bytes, we record the entropy (either on the fixed dataset or on a validation dataset if data is not fixed) and repeat the process. We stop when L has no collisions on the training data, and note that at each iteration of the algorithm we need only to look at data items which are not unique given previous bytes chosen for L, reducing algorithm runtime substantially (items that are not equal on a subset of bytes cannot be equal on a larger subset). At the end, we have a sequence of partial-key functions which are our solutions for $l = 1, 2, 3, \ldots$ bytes, with higher l meaning more input bytes are read but also monotonically increasing the entropy of the output.

Algorithm 1 ChooseBytes

Input: `CAO8=_3OCO`: either data items or sample of past data items
Input: `C4BC_3OCO`: data to check entropy on (if not for fixed dataset)

```
positions = vector()
entropies = vector()
max_len = maximum length of any data item
while not all partial keys unique do
    positions.push_back(NextByte(data,max_len,positions))
    entropies.push_back(EstimateEntropy(test_data, positions))
    data = NonUnique(data, positions)
return positions, entropies
```

Algorithm 2 NextByte

Input: `3OCO`: either data items or sample of past data items
Input: `<OG_;4=`: maximum length item in dataset
Input: `?OBC_1~C4B`: past bytes chosen

```
min_coll, min_i = 1, 1 // track of min # collisions, most entropic byte
for i = 0 to max_len do
    count_table, num_coll = {}, 0
    for j = 0 to len(data) do
        p_key = 3OCO»j¼ using (past_bytes, i) // form partial-key
        p_key = (len(data[j]), p_key) // length is always part of partial-key
        count_table[p_key] += 1 // increment count partial-key
        num_coll += (count_table[p_key] - 1) // add collisions (if any)
        if num_coll < min_coll then
            min_coll, min_i = num_coll, i // update best byte
return min_coll, min_i
```

Algorithms 1 and 2 give (simplified) pseudocode for this procedure. Additionally, Figure 4 shows example output from the procedure. While for simplicity Algorithm 1 is shown choosing 1 byte at a time, our implementation chooses 4 or 8 bytes at a time. This is because most modern hash functions which come after L operate one word of data at a time. In addition, we limit the maximum byte being chosen for partial-key hashing so that 90% of data items are under that data size. In the end, H looks as follows:

```
if len(x) > last byte used in L:
    return H(L(x))
else
    return H(x)
```

Because we designed L so that almost all keys satisfy the first if statement, this makes the full hash function have predictable branching statements. This initial if statement is also dropped if the keys are of fixed length. The result, when L is tightly integrated into the hash function H, is that H has predictable branches and a small instruction count on average.

Evaluating the Resulting Entropy. To make decisions on how many bytes are needed, we need an estimate of the entropy of $L(X)$. When data is fixed, we use the training set as a ground truth value for the entropy. When generalization to new data is needed, we use separate validation data.

To estimate the entropy of $L(X)$, we compute the empirical collision probability on the validation set V by 1) computing $L(x)$ for each x in V, 2) counting the number of collisions, and then

3) dividing this by $\frac{E^2}{2}$ where E is the number of items in $+$. From Lemma 1, this gives an unbiased estimate of the collision probability. To get an estimate $\hat{}_2$ of the entropy, we take the negative log of this number.

Given this estimator, the natural question to ask is "how many samples are needed?". The techniques of [4, 46] use the birthday paradox to answer this question; namely, if we want to say that the entropy is at least some value $_2$ with confidence, we need $\$^1 2\ ^{2\bullet 2o}$ samples. As we will show in Section 4, data structures or algorithms storing $=$ elements will generally need $_2$ to grow at a rate of $\log_2 =$, suggesting $\$^{1=1\bullet 2o}$ samples is enough to say with probability approaching 1 whether or not $!^{1-o}$ has enough entropy for a given task. Giving a concrete example, when using E validation samples a 99% confidence estimator for the entropy is: $_2 \quad \min\left(\hat{}_2, \log_2 \frac{E^2}{40}\right)$ with probability 0″99. Thus if our data structure needs entropy $_2 = \log_2 =$, setting $E \ ¡ \ \sqrt{40=}$ is enough validation samples to say with high probability whether or not $!^{1-o}$ has the required entropy. More details can be seen on this analysis in the technical report [29].

The most important takeaway is the fact that the number of validation samples needed both varies with the data size and also grows slowly with the data size. Thus, when we want to use Entropy-Learned hashing on small data, the sample can be small because we only need to make sure it has just enough entropy. When the data is large, the number of samples needed grows but much more slowly than the data size.

4 CONNECTING ENTROPY TO DATA STRUCTURE PERFORMANCE

The next step in Entropy-Learned Hashing is understanding the entropy needed for a given system task, i.e., a data structure or algorithm used in a system. As Figure 1 shows, hashing is used in a range of diverse systems to implement data structures and algorithms for various complex operations. We study specifically the entropy needed by three of the mostly widely used tasks, namely:

(1) **Hash tables** which are the default way to access data by equality, and which are widely used across general purpose programs including relational systems and key-value stores.
(2) **Bloom filters** which are used to reduce accesses to a set and are used in databases to reduce the costs of joins in OLAP systems as well as point queries in key-value stores.
(3) **Partitioning** which is a core step in numerous algorithms.

Each of these tasks has multiple metrics of interest, including: CPU cost, memory footprint, throughput, false positive rate, and much more. The three hash-based operations above present a diverse set of expressions of these metrics. For example, Bloom filters have small memory footprint compared to the other components, while they all have drastically different characteristics in terms of output write patterns which affects the overall throughput.

By creating cheaper to compute hash functions we improve the computational efficiency; what is left to show is that the small increase in expected collision probability does not result in significant degradation on other metrics. For hash tables, the metric of interest for performance is the number of comparisons needed to retrieve a key. For Bloom filters, it is the false positive rate and for Partitioning the variance of the distribution of data amongst bins.

There are two takeaways from the analysis in this section. The first is that we can argue formally about the needed entropy from partial-keys for data structures to behave as desired. This allows us to design Entropy-Learned hash functions which bring end-to-end performance benefits. Second, the analysis shows that across all tasks, Hash tables, Bloom filters, and Partitioning, the needed amount of Renyi entropy in $!^{1-o}$ is approximately $\log_2 =$ plus a constant 2. Thus, for a fixed dataset size, hashing needs only a constant number of bytes for enough uniformity in output and can be independent of key size. Additionally, the dependence on $=$ reaffirms our central thesis and further clarifies where Entropy-Learned Hashing is most useful: for large (hence random) objects or small datasets state-of-the-art hash functions do more work than necessary. The value of 2 depends on how much collisions affect a data structure; for instance, hash collisions in Bloom filters produce a certain false positive and so this has a high value of 2, whereas for hash tables a collision produces an extra comparison which is more tolerable and so 2 is lower.

4.1 Hash Tables

Two prototypical designs of hash tables are separate chaining and linear probing [20]. Separate chaining stores an array of linked lists. To query for an item, separate chaining hash tables 1) perform a hash calculation to get a slot 0 between 0 and $<\ 1$ and then 2) traverse the linked list at slot 0 until either the key is found or the end of the list is reached (the key is not present). Linear probing stores an array of keys and queries the table by 1) performing a hash calculation to get an initial slot 0, and then 2) traversing the array in sequential order until either the key is found, or until an empty slot is found (the key is not present). Separate chaining tables are easier to manage and analyze because collisions only matter for the same slot, however they have poor data locality because of many pointer traversals and require extra space for the many pointers. In contrast, linear probing offers better performance but is more difficult to analyze and manage because of complex dependencies between hash values.

4.1.1 Hash Tables: Separate Chaining.

Fixed Data. We first analyze separate chaining hash tables when the data is known which is an important class of indexed data. We then show this analysis translates from known data to random data.

Given $(_{j!} = {}^1\ _{j!}\bullet I^o$, when querying for an item ~ not in , the expected number of comparisons $\%^0$ is

$$E»\%^0 j\sim¼ = I_\sim \ , \ \frac{= I_\sim}{<} \quad I_\sim \ , \ U$$

This is because the (likely 0) $I_\sim$ items which have the same partial key for sure are in the same slot, and the other $= \ I_\sim$ items have $1\bullet <$ chance of being in the same slot. This cost of querying for a missing key is also equal to the cost of adding a new item into the hash table, and this relationship holds true for linear probing as well. This is because additions first verify the item is missing and then put the item into the first empty slot they find.

By the same logic, querying for a key G in costs $1 \ , \ \frac{1}{2}{}^1 I_G$ $1 \ , \ \frac{= I_G}{<}{}^o$ comparisons on average. The leading 1 is because the

query key for sure compares with itself, and the second term is 1•2 times the expected number of items in the same slot as G. Summing across all data, the average cost % of querying for a key satisfies:

$$E»\%¼ \quad 1 \, , \, \frac{1}{2}U \, , \, \frac{1}{2} \tilde{O}_{G2 \; j!} \frac{I_G^2}{=}$$

Random Data. When generalizing partial-key hashing to unseen (random) data, the above equations can be viewed as conditional expectations where we condition on the data. By using Adam's Law, i.e. »-¼= » »-j.¼¼, we can average over the possible produced datasets given by the random data. Using the union bound and Lemma 1, the expected cost of querying for a missing key and the average cost for querying for a key satisfy

$$E»\%^0¼ \quad U \, , =2^{\,2^{1!1-oo}} \tag{1}$$

$$E»\%¼ \quad 1 \, , \, \frac{1}{2}U \, , \, \frac{1}{2}{}^1= \quad 1^o2^{\,2^{1!1-oo}} \tag{2}$$

Comparison with Full-Key Hashing. For full key hashing, the corresponding costs for querying for a missing key and the average cost to query for a key are

$$E»\%^0¼ = U$$

$$E»\%¼ = 1 \, , \, \frac{1}{2}\frac{=}{<} \quad 1 \, , \, \frac{1}{2}U$$

This shows the tradeoff between partial key hashing and full key hashing. The number of comparisons is lower for full-key hashing, but this advantage goes exponentially fast to 0 as the entropy of the partial key hash increases. At the same time, the partial-key hash is significantly cheaper to compute.

Looking at the required relationship between = and the needed entropy of the input sub-keys further clarifies when and why partial-key hashing is useful. When $2^{1!1-oo} \; i \; \log_2 =$, the number of extra comparisons needed drops below 1 and continues to drop exponentially fast with more entropy. Since hashing objects is more expensive than comparing them, this point represents near definite savings; the hash computation for the table is much faster while the work after the hash function is nearly the same.

4.1.2 Hash Tables: Linear Probing.

Because of the complex dependencies between hash values and collisions, linear probing is significantly more complicated to analyze resulting in lengthier proofs. We provide a high level overview of the results while all detailed proofs can be found at the technical report [29]. We start with full-key hashing. We analyze the expected length of a full chain) for a new item added to the hash table. The chain includes the empty position on a chain's right side but not on its left side. Figure 3 shows an example.

Full-Key Hashing. In the technical report [29], we provide a novel analysis of linear probing showing that the expected length of) satisfies $»)¼ = \&_1{}^1{<}•{=}^o$ where $G^{\underline{:}}$ is the : -th falling power and $\&_8{}^1{<}•{=}^o = \sum_{:=0}^{8} \frac{=^{\underline{:}}}{<^:}$. For a new item, each location in a probe chain is equally likely as a hash location and so the expected probe cost given) is $»\%^0 j) ¼ = \frac{1}{2} \, , \, \frac{1}{2})$. Using Adam's law, it follows that

$$»\%^0¼ = \frac{1}{2}{}^1 1 \, , \, \&_1{}^1{<}•{=}^{oo} \quad \frac{1}{2}{}^1 1 \, , \, \frac{1}{1 \quad U^{o2}}{}^o$$

which matches the known equations given by Knuth in [32].

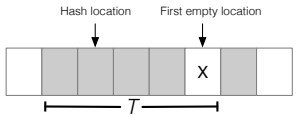

Figure 3: Example of a linear probing chain.

The average cost to query a key is then equal to the average cost to insert each key. Since the insertion cost $»\%^0¼$ depends on =, we use $\%_8^0$ to denote the cost when there are 8 keys in the table. The average cost to query a key is then

$$»\%¼ = \frac{1}{=} \tilde{O}^{=-1}_{8=0} »\%_8^0¼ = \frac{1}{2}{}^1 1 \, , \, \&_0{}^1{<}•{=} \quad 1^{oo} \quad \frac{1}{2}{}^1 1 \, , \, \frac{1}{1 \quad U}{}^o$$

Partial-Key Hashing: Fixed Data. When given $(_{j!} = {}^1{}_{j!}•I^o$, the expected length of the probe chain) depends on the number of partial key matches for the inserted key ~, and satisfies

$$») ¼ \quad \&_1{}^1{<}•{=}^o \, , \, I_{\sim}\&_0{}^1{<}•{=}^o \, , \, \tilde{O}_{G<\sim} \frac{I_G^2}{<}\&_1{}^1{<}•{=}^o$$

$$\frac{1}{{}^1 1 \quad U^{o2}} \, , \, \frac{I_{\sim}}{1 \quad U} \, , \, \tilde{O}_{G<\sim} \frac{I_G^2}{<{}^1 1 \quad U^{o2}}$$

When the new key is unique, the most common scenario when $2^{1!1-oo}$ is high, each location in the probe chain is equally likely and so $»\%^0 j) ¼ = \frac{1}{2} \, , \, \frac{1}{2})$. However, when the new key is not unique, each position in the chain is no longer equally likely. Thus we make the worst case assumption that it is at the end of the probe chain.

$$»\%^0¼ \; \begin{cases} \frac{1}{2} \, 1 \, , \, \frac{1}{{}^1 1 \quad U^{o2}} \, , \, \sum_{G<\sim} \frac{I_G^2}{<{}^1 1 \quad U^{o2}} & \text{if } I_{\sim} = 0 \\ \frac{I_{\sim}}{1 \quad U} \, , \, \frac{1}{{}^1 1 \quad U^{o2}} \, , \, \sum_{G<\sim} \frac{I_G^2}{<{}^1 1 \quad U^{o2}} & \text{if } I_{\sim} \; i \; 0 \end{cases} \tag{3}$$

When translating from $\%^0$ to %, we again have that $»\%¼ = \sum_{8=0}^{=-1} »\%_8^0¼$. Since the cost of inserting each key is no longer the same, there is the question of how to evaluate this expression. Here, we make use of a fact first noticed in [52], that the average cost of querying is equal for any order in which the items are inserted. Thus, in evaluating $»\%¼ = \sum_{8=0}^{=-1} »\%_8^0¼$, we may choose the insertion order of the items. Inserting all keys with non-unique partial-keys first and then inserting all keys with unique partial-keys gives the following bound for »%¼.

$$\begin{aligned} »\%¼ \quad & \frac{=-3}{2=} \, , \, \frac{1}{2}\&_0{}^1{<}•{=}^o \, , \, \frac{2}{<}\&_0{}^1{<}•{=}^o \, , \, \frac{2 \, , \, 3}{2=}\&_0{}^1{<}•3^o \\ & \frac{1}{2}{}^1 1 \, , \, \frac{1}{1 \quad U}{}^o \, , \, \frac{2}{=} \, , \, \frac{2}{<}\frac{1}{1 \quad U} \\ & {}^1\frac{1}{2} \, , \, \frac{2}{=}{}^o{}_1 1 \, , \, \frac{1}{1 \quad U}{}^o \end{aligned} \tag{4}$$

We use $2 = \sum_G I_G^2$ for the number of collisions and $3 = \sum_{G:I_G \; 2} I_G$ as the number of items that are duplicated keys. The above approximation assumes that 3•< is small, which is the case whenever most keys are unique. This holds true with probability near 1 if entropy is sufficiently large.

Random Data. Using equations (3), (4), and Lemma 1 as well as Adam's Law, we have

$$\%'^{1}_{4} \quad \frac{1}{2}^{1}1 \, , \, \frac{1}{^{1}1 \quad U^{o}2}^{o} \, , =2 \quad {}^{2^{1}!^{1}-^{oo}} \frac{3}{2^{1}1 \quad U^{o}2} \tag{5}$$

$$\%'^{1}_{4} \quad \frac{1}{2}^{1}1 \, , \, \frac{1}{1 \quad U}^{o} \, , =2 \quad {}^{2^{1}!^{1}-^{oo}}1 \, , \, \frac{1}{1 \quad U}^{o} \tag{6}$$

Comparison With Full-Key Hashing. The tradeoffs between partial-key hashing and full-key hashing are similar to separate chaining. Again, we have a slight increase in comparisons as a trade-off for significantly faster hash function evaluation. The expected number of comparisons again drops exponentially fast with the source entropy and $2^{1!^{1}-^{oo}}$ needs only to be in the same order of magnitude as $\log_2{}^{1}=^{o}$ for the extra needed comparisons to be small. Thus, as before, partial-key hashing makes the work of computing hash functions significantly cheaper while the work after the hash function is near identical, producing a net performance benefit.

4.2 Bloom Filters

For Bloom filters, the central trade-off is between the speed of the filter and the false positive rate (FPR) of the filter. As the number of bytes given as input to the hash becomes smaller, hashing becomes faster but there is a greater possibility of a partial-key collision, creating a certain false positive.

More formally, let $\%'^{1}<\bullet=\bullet \ ^{o}$ denote the false positive rate of a Bloom Filter using $<$ bits, storing $=$ items and using a hash function . For a Bloom Filter using partial-key hash $^{0} = \ !$, its number of set bits is a function of the number of distinct items fed to . If no keys collide on $!$, then it becomes a traditional Bloom Filter storing $=$ items and using . If there are $=^{0} \ddot{Y} =$ distinct items after $!$, then the resulting filter structure has the same number of set bits as one containing $=^{0}$ items. So for query key $\sim 8 \ !$, it has a false positive rate of $\%'^{1}<\bullet=^{0}\bullet \ ^{o}$, whereas if $\sim 2 \ !$ it has a false positive rate of 1. It follows that our Bloom Filter using ^{0} has exactly the following false positive rate:

$$\%'^{1}<\bullet=\bullet \ ^{0o} = \mathrm{P}^{1}.\mathrm{j}! \ 2 \ !^{o} \, , \ \%'^{1}<\bullet=^{0}\bullet \ ^{o} \tag{7}$$

The second term is less than $\%'^{1}<\bullet=\bullet \ ^{o}$ as Bloom Filters' false positive rates increase with the number of items stored. If keys and non-keys are very different conditioned on the set of bytes $!$, then it is possible to make the FPR less than that of a standard Bloom filter by having $=^{0} \ddot{Y}\ddot{Y} =$ and $\mathrm{P}^{1}.\mathrm{j}! \ 2 \ !^{o} \quad 0$. However, we will generally ignore this case and focus on the case where keys and non-keys have the same distribution conditioned on $!$. In this case, a convenient bound for (7) is

$$\%'^{1}<\bullet=\bullet \ ^{0o} \quad \mathrm{P}^{1}.\mathrm{j}! \ 2 \ !^{o} \, , \ \%'^{1}<\bullet=\bullet \ ^{o} \tag{8}$$

which is the FPR of a standard Bloom filter plus the probability that the query key matches some item in the key set on the bytes $!$.

Using the union bound, equation (8) translates to:

$$\%'^{1}<\bullet=\bullet \ ^{0o} \quad =2 \quad {}^{2^{1}!^{1}-^{oo}} \, , \ \%'^{1}<\bullet=\bullet \ ^{o} \tag{9}$$

Comparison With Full-Key Hashing. The above analysis reaffirms the central takeaway of our analysis of hash tables; the entropy of the dataset needs to be on the order of $\log_2 =$. For Bloom filters, a reasonable additional goal is that the increase in FPR be no more than some chosen γ. In this case, we need $2^{1!^{1}-^{oo}} \ i \ \log_2 = \, , \, \log_2{}^{1}1\bullet\gamma^{o}$. So an additional entropy term is needed to say that collisions are very rare for new partial-keys. As we show in our experiments, datasets often have this surplus entropy and so the Bloom Filter becomes significantly faster without suffering any false positive rate increase.

4.3 Partitioning & Load Balancing

With Partitioning the goal is to distribute $=$ items, e.g., tuples or computational tasks, to a set of $<$ bins. Here, we characterize how even this allocation is by analyzing the variance of the number of items assigned to each bin when each input key is unique. At lower variances, each bin is distributed closely around the average number of items $=\bullet<$ whereas higher variance suggests the bins are highly uneven. One important challenge comes when keys are skewed and heavy hitters exist. While challenging, the unevenness comes from the existence of heavy hitters rather than the quality of the hash function, and so we focus on the hash quality by considering the partitioning of all unique items.

Full-Key Hashing. With full-key hashing, the variance of each bin is the variance of a binomial with $=$ balls each with probability $1\bullet<$. Thus for a specific bin, its number of assigned objects $.$ has $+\mathit{OA}^{1}.^{o} = \frac{=}{<} \quad \frac{=}{<^{2}}$.

Partial-Key Hashing: Fixed Data. The probability of each key in $!$ being assigned to a specific bin is distributed as an independent Bernoulli trial with probability $\frac{1}{<}$. Letting $\mathbf{1} \ {}_{1G^{o}=8}$ be the event that G was hashed to bin 8, the variance of the number of objects $.$ assigned to bin 8 is

$$+\mathit{OA}^{1}.\mathrm{j}_{!}{}^{o} = +\mathit{OA}^{1} \tilde{O}_{G2 \ \mathrm{j}!} \ I_{G}\mathbf{1} \ {}_{1G^{o}=8}{}^{o} = {}^{1}= \, , \tilde{O}_{G2 \ \mathrm{j}!} \ I\frac{2}{G}{}^{o}1\frac{1}{<} \quad \frac{1}{<^{2}}{}^{o}$$

Partial-Key Hashing: Random Data. For random data, we use the same conditioning arguments as before. Using Eve's Law, i.e. $+\mathit{OA}^{1}.^{o} = \mathrm{E}\text{»}+\mathit{OA}^{1}.\mathrm{j}_{!}{}^{o}{}_{4} \, , \, +\mathit{OA}^{1}\mathrm{E}\text{».}\mathrm{j}_{!}{}_{4}{}^{o}$, we can calculate the variance on random data. First, we note that for any set $!$, the value of $\text{».}\mathrm{j}_{!}{}_{4}$ is $=\bullet<$ by the randomness of the hash function (each bin is equally likely to contain any item). Thus $+\mathit{OA}^{1}\mathrm{E}\text{».}\mathrm{j}_{!}{}_{4}{}^{o} = 0$ and again using Lemma 1, we have

$$+\mathit{OA}^{1}.^{o} \quad {}^{1}1 \, , =2 \quad {}^{2^{1}!^{1}-^{oo}}{}^{o}1\frac{=}{<} \quad \frac{=}{<^{2}}{}^{o} \tag{10}$$

Comparison With Full-Key Hashing. As before, $2 \ i \ \log_2 =$ is enough for partial-key hashing to have similar variance to full-key hashing in terms of absolute terms. Thus, as in prior cases, once $2 \ i \ \log_2 =$ we have faster computation in terms of partitioning without sacrificing on the quality of our partitioning.

An important secondary argument for load balancing is whether we care about the absolute deviation from the mean or the percentage deviation away from the mean. While the absolute variance grows with $=$, the relative standard deviation, i.e. the standard deviation over the mean, of the bins decreases with $=$ so that it becomes less and less likely that some bin has $G\%$ more than its expectation. In particular, the relative standard deviation is less than

$$\sqrt{\frac{<}{=}}\sqrt{\frac{}{1 \, , =2 \quad {}^{2^{1}!^{1}-^{oo}}}} \quad \sqrt{\frac{}{<2 \quad {}^{2^{1}!^{1}-^{oo}}}} \tag{11}$$

Since the expected distance from the mean for a binomial is dominated by its standard deviation [12], the above statement actually

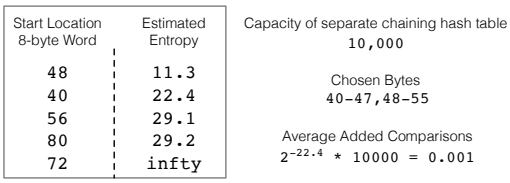

Figure 4: The amount of bytes needed is based on the data and the current data structure capacity.

says that a bin's expected proportional deviation away from its mean is less than (11). So for instance, if we want a partition to be within 5% of its mean on average, we can achieve this by having $2 \log_2 \frac{1}{0.05} \geq \log_2 <$.

Thus partitioning and load balancing have two regimes with regards to Entropy-Learned Hashing. When small absolute variance is required, we need $2^{-H(A)} \geq \log_2 =$; however, when = is large and we are simply interested in bins being relatively similar sizes, we can let $2^{-H(A)}$ be greater than $\log_2 <$ plus a small constant, where the constant controls how much deviation is allowed.

5 RUNTIME INFRASTRUCTURE

Section 3 showed how to estimate the entropy of datasets when conditioned on partial-keys and Section 4 showed how much entropy is needed for important hashing-based tasks. This section brings everything together by explaining how to utilize Entropy-Learned Hashing at run time: namely, given a hash-based task and analysis of a dataset, choose the Entropy-Learned Hash function to have just enough randomness. Additionally, this section covers runtime infrastructure related to robustness so that Entropy-Learned Hashing retains the trustworthiness of traditional hash data structures.

Creating Hash Tables. Hash tables have a maximum capacity beyond which they need to rehash the stored items into a new larger table. This keeps the load factor low and therefore query times low. For Entropy-Learned Hashing, we use this maximum capacity before rehashing to decide $!$. In particular, for separate chaining hash tables, we choose $!$ such that $2^{-H(A)} \geq \log_2 = \geq 1$, where = is the maximum number of items the current table will hold before rehashing. For linear probing hash tables, we choose $!$ so that $2^{-H(A)} \geq \log_2 = \geq \log_2 5$. Both values are chosen based on the equations governing the number of comparisons, i.e. equations (1), (2), (3), and (4), and make sure the number of comparisons executed using partial-key hashing and full-key hashing are similar. An example of how the current capacity is used to choose $!$ is shown in Figure 4, where an initial table with capacity 1000 uses just the 8-byte word at location 48 to hash keys.

As the capacity of a hash table changes (as new items are inserted), a rehash is triggered causing each item to be reinserted. Entropy-Learned Hash tables uses this opportunity to change the hash function; for instance, when key 1001 is inserted into the hash table from Figure 4, a rehash is triggered causing the table to grow. If the new capacity is above $2^{11.3} = 2521$, the partial-key function adds another word to increase entropy to the required amount. As a result, the hash table maintains just the right amount of entropy needed throughout its life cycle, using as cheap a hash function as possible without adding substantial extra collisions.

Bloom Filters. Bloom Filters need an estimate on the number of items they will hold before their creation. This is because, without access to their base items, they have no access to grow the number of bits being used. While there are techniques around this [5], these come with space and computation tradeoffs and it remains true that standard Bloom filters need an up-front estimate of the number of data items. For Entropy-Learned Hashing, this makes it simple to choose the hash function. Given a maximum number of items = and an allowable added FPR of γ, we set the partial-key hash function to have entropy $2^{-H(A)} \geq \log_2 = \geq \log_2 1 \bullet \gamma$.

Partitioning. For partitioning we require an estimate on the maximum number of items to be partitioned. We also need user input on how even they want partitions to be. If absolute variance is of primary importance (so that partitions are unlikely to vary by more than some # of tuples regardless of partition size), then setting $2^{-H(A)} \geq \log_2 = \geq 2$ assures that variance is no more than $1 + 2^{-2}$ times its usual amount. The default value of 2 which we use is 3. When relative variance is more important, and users need partitions to be roughly even (i.e. within 100c% of each other's size), we set $2^{-H(A)} \geq \log_2 < 2 \log_2 2$ as dictated by equation (11). We use $2 = 0.05$ by default so that partitions are expected to be within 5% of their expected size.

Robustness. While Entropy-Learned Hashing makes only weak assumptions, namely that data which are somewhat random remain somewhat random, it recovers good performance quickly when assumptions are violated. Entropy-Learned Hashing is the most robust for hash tables. This is for multiple reasons, namely: 1) if collisions are as expected on items in the dataset, queries for both keys in the data and not in the data return quickly (Section 4), 2) hash tables can monitor collisions during insertions with little overhead, and 3) rehashing is an acceptable operation in hash tables by default (it occurs in all standard hash table libraries). This third point is the most key, and Entropy-Learned Hashing can rehash hash tables if collisions ever deviate from what is expected, falling back to full-key hashing if needed. For Bloom filters, their # of set bits concentrates sharply around their expected value [14], and this fact is used during construction of Entropy-Learned Bloom filters to validate that the data items fit the expected level of randomness. However, if they do not, or if queries are substantially different than the inserted items, the filter must be rebuilt. For partitioning, the cost of overloaded bins depends on the context, but for many contexts, such as in-memory radix partitioning, this can be solved by dividing the one or two overloaded bins into multiple bins. Section 5 of the technical report covers robustness in more detail.

6 EXPERIMENTAL EVALUATION

We now demonstrate that, by identifying and utilizing surplus randomness in data, Entropy-Learned Hashing brings critical performance benefits against the top hash functions used at scale today by Google and Meta and across a diverse set of hash-based core components of modern systems.

Our experimental evaluation consists of 3 parts. The first part, which contains the bulk of our experiments, shows that Entropy-Learned Hashing produces sizable benefits of up to 3.7 , 4.0 , and 14 for common medium-sized key types such as URLs and text data. The second part of our experimental section covers benefits from Entropy-Learned Hashing on large keys such as those that would appear in deduplicating file blocks, with speedups of several

orders of magnitude. Finally, we cover training time for Entropy-Learned Hashing and present the run times for applying the greedy algorithm to select bytes to hash.

6.1 Setup and Methodology

Data Structures and Operations. We use a diverse set of data structures and operations to apply Entropy-Learned Hashing: we test with Hash tables, Bloom filters, and Partitioning.

For **hash tables**, we compare against Google's hardware-efficient linear-probing hash table implementation, SwissTable [27, 36]. This is the default hash table used in C++ throughout all Google operations, and has been heavily optimized as a result of the large computational footprint of hash tables at Google. A particular implementation note for SwissTable is that it first does linear probing into an array of tag bits (8 bits per key) to see if chosen bits from hash values match, and only if they do, compares the full items. This means probing for keys not in the table is cheaper than probing for keys stored in the table. We also compared against F14, the default hash table used at Facebook [16]. The results are extremely similar and so we include only results with SwissTable.

For **Bloom Filters**, we implemented register blocked Bloom filters from [37]. To cut down hashing time, and thus to be conservative with respect to our benefits, we used a variant of double hashing wherein we compute one 64 bit hash function, split it into two 32-bit hash values, and then use these as the inputs to double hashing [31]. We also utilize the techniques for fast modulo reduction by multiplication from [61].

For **partitioning**, many of the techniques devised by database research such as software write buffers [67] and non-temporal stores [10] do not apply to large data types or variable length data types. Thus our partitioning is a simple for loop that computes hash values and writes out data directly to a partition.

Base Hash Functions. We use three state-of-the-art hash functions. For hash tables, we use wyhash, which is one of the two default options used in SwissTable. We use both the version contained in SwissTable as well as the most recent optimized version of wyhash given directly by the author [66]. For Bloom filters we use xxh3, which is used widely at Facebook and is the default for the Bloom filters in RocksDB [19]. For partitioning we use the implementation of CRC32 from the OLAP database Clickhouse [70].

Implementation. We modify each of the three base hash functions. We maintain their basic interface (input is an array of bytes plus a key length), and tightly integrate Entropy-Learned Hashing. Thus there is Entropy-Learned xxh3, Entropy-Learned wyhash, and Entropy-Learned CRC32. The bytes chosen to hash are selected at hash function construction and stored in a const array. The functions read from 30C0»;>20C8>=B»8¼¼ instead of 30C0»8¼, and we use templates to generate efficient code for partial-key hash functions using 1,2,3,4,.. words. These templates modify the initial function to reduce branching statements because of the known length of the partial-key. All implementation is in C++. All experiments for hash-based tasks are in-memory since hash-based tasks typically run in-memory. For example, a hash table should always fit in memory to get good performance while a Bloom filter will also typically reside in memory to protect from expensive disk access. Thus such structures are both created and utilized in memory. When disk is involved, the CPU cost of hashing is typically not highly visible in terms of operational latency unless very fast disk devices such as SSDs are used (although CPU usage is still reduced).

Processor	Intel Xeon E7-4820 v2
#sockets	4
#cores per socket	8
Hyper-threading	2-way
Turbo-boost	Off
Clock speed	2.00GHz
L1I / L1D (per core)	32KB / 32KB
L2 (per core)	256KB
L3 (shared)	16MB
Memory	1TB

Table 2: Server Parameters

Dataset name	Avg. key length	# keys
UUID	36	100K
Wikipedia	129	22K
Wiki	22	99K
HN URLs	75	247K
Google URLs	81	1.2M

Table 3: Real-world data.

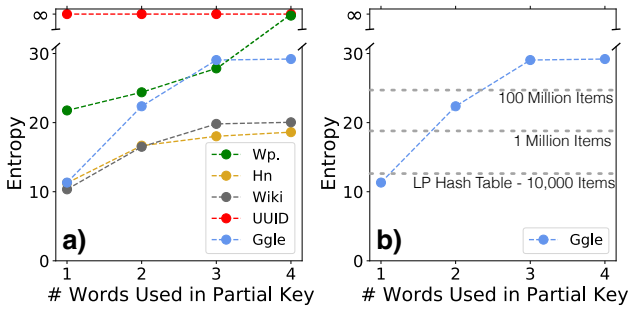

Figure 5: The entropy of a dataset grows quickly with the amount of words being hashed. By 4 words, most datasets support data structures with millions of elements.

Datasets. We use five real-world datasets for experimentation. Two datasets consist of URLs, with one containing the URLs of stored Google Landmarks and the other all URLs posted to Hacker News during 2015 [2, 45]. The other three, UUID, Wikipedia, and Wiki, are database columns taken from a recent research study [13]. They contain universally unique identifiers, sampled text from Wikipedia, and Wikipedia entry titles respectively. Table 3 presents the number of items and average key length for each real-world dataset. In addition, we use synthetic data to have finer control over key size and data size. Section 6.3 uses 80 byte keys with bytes 32-39 drawn randomly from the alphabet (26 possible values), and all other bytes constant. Section 6.6 uses 8KB keys with each byte ideally random.

Experimental Setup. We use an Intel Ivy Bridge server. Table 2 summarizes the server parameters. We use Debian GNU/Linux 10 operating system. Data structures are queried for a warmup phase before timing and input keys for queries are in cache. We pin the thread to a particular core and locally allocate memory. We use Intel VTune's uarch-exploration [30] for performing hardware-level time breakdown and Linux perf [39] for performing memory-level parallelism tests and software-level time breakdown.

6.2 Number of Words vs. Entropy

Before demonstrating performance results, we first make the idea of surplus randomness more concrete with examples from real data. We show that for many datasets with medium-sized keys, good hashing properties can be achieved for data structures with millions of elements while hashing only parts of the keys. We divide each dataset in Table 3 in half. We use the first part to choose which bytes to hash in a greedy manner as described in Section 3. This

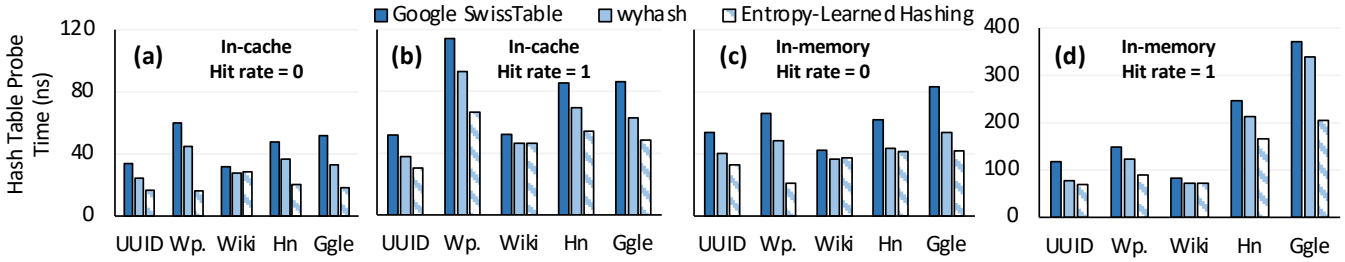

Figure 6: Entropy-Learned Hashing reduces probe times for hash tables across datasets, data sizes, and hit rates.

produces an ordered list of bytes (or words) to choose. Choosing more bytes from the list produces a partial-key function providing more entropy. We use the second half of the dataset to get an unbiased estimate of the entropy for each combination of bytes as described in Section 3. Figure 5a shows that the entropy of the result of the partial-key function increases for all datasets with the number of words included. We see that by 3 words being included all datasets have an entropy of at least 18, and 3 of the 5 have entropies above 25. For Wikipedia and UUID, infinite entropy is estimated because no collisions are observed with the partial-key function. Figure 5b shows how this entropy translates into data structures, where we see that the Google URLs dataset is capable of using partial-key hashing with hundreds of millions of elements while hashing just a couple words. Similar results can be seen by transposing the other four datasets onto Figure 5b, with most datasets supporting hash data structures larger than the actual number of elements found in the dataset.

6.3 Hash Table Probe Time

After showing that datasets have enough entropy for partial-key hashing to be used,we now turn to showing the performance benefits which can be gained by using Entropy-Learned hash functions for data structures and algorithms. We first focus on hash tables. We examine the probe time per hash table lookup, where we perform the lookups one after the other without any blocking, e.g., similar to the probe-phase of the hash join algorithm.

Entropy-Learned Hashing Reduces Hash Table Probe Time. We first test hash table probe times on real-world datasets for small (L1-resident) and large data (L3/DRAM-resident) with 0% (hit rate = 0) and 100% (hit rate = 1) hit rates. We test with Google's SwissTable using three hash functions: (i) the default hash function provided by SwissTable (GST), (ii) the most recent version of wyhash (FK), and (iii) the Entropy-Learned wyhash hash function (ELH). The small data contains one thousand keys, and the large data contains half of the number of keys of the dataset (we use the other half to generate probes for missing keys). Figure 6 shows the results, wherein Entropy-Learned Hashing provides speedups across all data sizes, datasets, and hit rates over full-key hashing. Across the 20 experiments, the average speedup using ELH over wyhash and SwissTable's default hash function is 1.40 , with these speedups being as high 3″7 over the default hash function of SwissTable and as high as 2″9 over wyhash, both of which are well engineered functions and implementations.

Entropy-Learned Hashing Scales with Entropy, not Key Size. To understand the reasons behind the speed up observed in Figure 6, we first need to return to Table 3 and Figure 5. For full-key hashing,

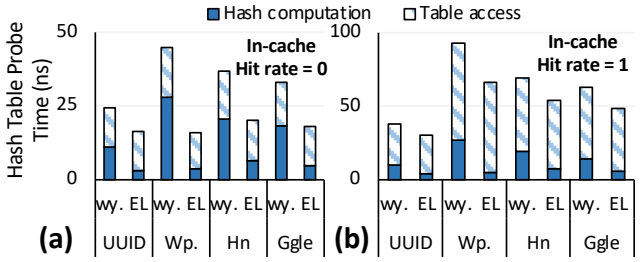

Figure 7: Entropy-Learned Hashing significantly reduces computation time bringing speedup as high as 2.9 for cache-resident hash tables with (a) low and (b) high hit rates.

it needs to hash each byte of the dataset, and so the number of bytes processed is on average the key length given in Table 3. For Entropy-Learned Hashing, the number of bytes it hashes is when the entropy of the dataset (seen in Figure 5a) crosses the entropy needed by the data structure (seen in Figure 5b). When there is a large gap between these two numbers, Entropy-Learned Hashing produces large speedups. For instance, the large gap between the number of bytes hashed is why ELH achieves 2.9 speedup over wyhash and 3.9 speedup over default SwissTable in Figure 6a. Similarly, it is why ELH is 1.67 faster than wyhash and 1.81 faster than default SwissTable on the Google dataset in Figure 6d.

While faster hashing computation uniformly brings speedups to hash table probes, the amount of this speedup depends on other factors of hash table queries, namely the hit rate and hash table size. We now explain how the combination of these factors with Entropy-Learned Hashing affects performance.

Computation Dominates for Cache-Resident Hash Tables. For cache-resident hash tables, memory requests return quickly and so computation dominates the overall cost of probes. In this case, the savings created by Entropy-Learned Hashing depend on how much work there is beyond the hash function evaluation. Figure 7 shows how the work beyond hashing differs for queries for non-existing keys and for existing keys. When queries are for non-existing keys, computation usually consists of the hash function plus small amounts of computation using the tag bits. As Figure 7a shows, in this case the hash function evaluation is most of the cost and Entropy-Learned Hashing brings significant benefits. This explains the 1.5 , 2.9 , 1.8 , and 1.8 speedup over wyhash seen in Figure 6a for the UUID, Wikipedia, Hacker News, and Google datasets, respectively. When queries are for keys in the dataset, Figure 7b shows the comparison after the hash function evaluation takes significant time. As a result, Entropy-Learned Hashing still provides benefits but not quite as large as before, with the savings being 1.23 , 1.41 1.28 , and 1.28 for the UUID, Wikipedia,

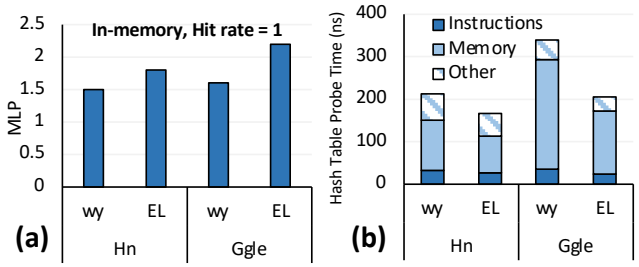

Figure 8: (a) MLP is significantly higher for ELH than it is for full-key hashing. (b) As a result, ELH reduces both the number of instructions executed and memory waiting time.

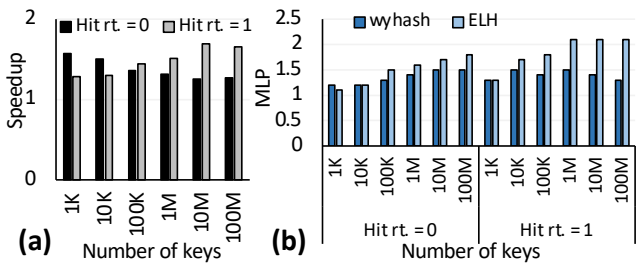

Figure 9: (a) Entropy-Learned Hashing provides larger benefits for missing keys at small data sizes and larger benefits for existing keys at large data sizes. (b) Entropy-Learned Hashing improves memory-level parallelism.

Hacker news, and Google datasets, respectively. Thus in cache, Entropy-Learned Hashing provides up to a 40% speedup for queries on existing keys and up to a 3 improvement on non-existent keys.

Memory Parallelism Dominates for Large Hash Tables. At large data sizes, the increase in computational performance from faster hashing leads to more efficient use of the memory hierarchy. This is due to the effects of CPU pipelining. Namely, when hash table lookups are done one after the other without blocking, then the CPU typically pipelines multiple hash table lookups which are then executed in parallel [34]. Entropy-Learned Hashing reduces the amount of computation required, and as a result, the CPU fits a larger number of hash table lookups into its pipeline. The effect of this increased pipelining is what creates the speedups seen at large data sizes in Figure 6c and 6d across datasets, with Entropy-Learned Hashing being as much as 1"67 faster than the nearest competitor.

The amount of this savings depends on the costs of memory accesses, with more expensive memory accesses leading to larger improvements. For instance, in Figure 6d we see that the larger datasets Google and Hacker News produce greater savings than the smaller datasets Wikipedia, UUID, and Wiki. Similarly, comparing Figure 6d to 6c, querying for existing keys produces greater savings because we view both tag bits and full-keys in comparison to just the tag bits most often for missing keys.

Figures 8a and 8b refine this analysis. Figure 8a shows the memory-level parallelism (MLP), which is defined as the number of L1 data cache misses per CPU cycle, for the Hacker News and Google datasets using hit rate = 1. The higher MLP in 8a indicates that a large number of data cache misses are being executed in parallel by Entropy-Learned Hashing than by full-key hashing. Figure 8b shows how this affects the overall runtime of hash table probes under the same setup, with Entropy-Learned Hashing reducing both the number of instructions executed and memory waiting time. This analysis corroborates the results seen in Figure 6c and 6d, where Entropy-Learned Hashing provides a 1.31 speedup on average over full-key hashing.

Entropy-Learned Hashing Scales with Data. We now turn to experiments with synthetic data so that we can more finely control the data size and experiment with larger data sizes. Figure 9a shows the main result, which is that Entropy-Learned Hashing provides benefits for hash tables across small and large data sizes. At small data sizes of 1K tuples, Entropy-Learned Hashing provides 2.33 speedups on queries for non-existing keys and 1.30 speedups for existing keys. For large data sizes of 100M tuples, this speedup is 1.3 for missing keys and 1.7 for existing keys. Figure 9b shows that the reason for these speedups is as discussed before for the real-world datasets. Namely, at small data sizes the savings in computation directly produce speedups for Entropy-Learned Hashing, whereas for large data sizes the more efficient hash computation leads to better MLP which produces faster probe times.

6.4 Bloom Filter Lookup Time & FPR

In this section, we evaluate Entropy-Learned Hashing for Bloom filters. We examine the lookup time and false positive rate (FPR) metrics. As input parameters, we let the FPR of the filter be 3% and allow the Entropy-Learned Hashing filter to deviate in FPR by 1%. The filter uses 3 hash functions, but computes only 1 due to double hashing. All parameters are tunable; this experimental setup is meant to reflect high-throughput filters such as those in filter push-down before joins [37]. For the small data size we use 1K keys and for the large data size we again use half the number of keys in the data.

Entropy-Learned Hashing Reduces Filter Lookup Time. Figures 10a and 10b present results for Bloom filters lookup time and FPR using xxHash and Entropy-Learned Hashing. Figure 10a shows that Entropy-Learned Hashing improves performance on high entropy datasets such as Google, Hacker News, UUID, and Wikipedia. The speedup is consistently between 1.85 and 4.51 . For Wiki, which has both small key size and low entropy, the speedup is small. Across all datasets, the average speedup is 2.10 , so that Entropy-Learned Hashing consistently provides drasticaly faster throughput on Bloom filter queries.

Entropy-Learned Hashing has Tunable Added FPR. Figure 10b presents the FPR of Bloom filters using Entropy-Learned Hashing and full-key hashing. Most importantly, as can be seen in Figure 10b, the FPR is within 1% as our tuning parameter suggests so that our analytical bounds hold. Additionally, Figure 10b shows that the increase in FPR is usually much less than this tuning parameter, in this case being only 0.1%. Thus, for most datasets the difference in FPR is negligible. Additionally, this FPR increase can be adjusted down or up as needed. Reducing the allowed increase in FPR increases the entropy needed and so requires more hash computation, and so this represents a tunable FPR vs. speed tradeoff.

Bloom Filters require more entropy than Hash Tables. For a dataset size of = and added FPR of γ, ELH requires $\log_2 =$ ¸ $\log_2{}^1 1 \bullet \gamma^0$ entropy, which is approximately $\log_2{}^1 1 \bullet \gamma^0$ more entropy than hash tables. For certain datasets such as Wiki or Hacker News, this goes beyond the entropy they can provide using small partial-keys and

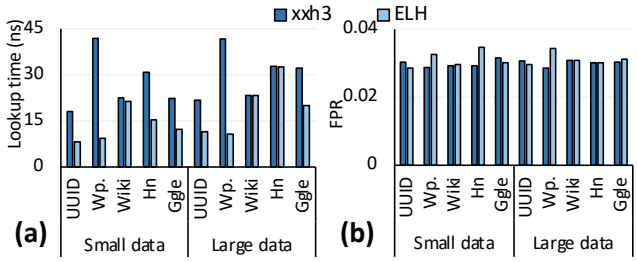

Figure 10: Improving Bloom filter lookup time (a) and false positive rates (b) for small and large data sizes.

	Pure hashing		Pos. id.		Data	
# Par.	64	1024	64	1024	64	1024
UUID	3.15	3.15	2.05	1.38	1.01	1.00
Wp.	14.10	14.09	6.18	2.66	1.23	1.18
Wiki	1.25	1.09	1.37	1.10	1.01	1.01
Hn	4.29	1.00	2.72	1.00	1.17	1.03
Ggle	7.83	7.82	2.51	1.42	1.01	1.00

Table 4: Speeding up when partitioning.

so they revert to using full-key hashing at large data sizes as can be seen in Figures 10a and b. For Google URLs, Wikipedia, and UUID, they have more than enough entropy and each can support at least 100 more data or a 100 lower added FPR. Thus, these datasets maintain consistent speedups at no cost to FPR for very large data sizes as seen in Figure 10b.

6.5 Partitioning Time & Variance

Partitioning is used in many contexts. For instance, tuples may be sent across the network in settings such as map-reduce or simply partitioned in memory as in radix-partitioning before hash joins. Because of this, the cost of partitioning depends very heavily on the application it is used in. To help guide users in terms of whether Entropy-Learned Hashing can be useful for their application, we provide three micro-benchmarks. These benchmarks show the increased computational efficiency of Entropy-Learned Hashing on partitioning and put this computational efficiency in context. In the first micro-benchmark, we only compute the partition assigned to each input key. In the second, we keep a list of positional identifiers for each partition and write out the position of each key assigned to each partition. In the third, we write out the actual keys assigned to each partition. As we progress through the microbenchmarks, we move from a computationally heavy task with few writes to a memory bandwidth intensive task which is mostly memory bound. Depending on the setup, the benefit in performance from using Entropy-Learned Hashing may be between 14 and 18%. Thus, the benefit of Entropy-Learned Hashing for partitioning depends on whether the saved computational cycles are of use, either directly through speedups on the task at hand, or indirectly, by allowing other computation to take place while network or memory I/O is being performed. Like Bloom Filters, partitioning has a tunable parameter which allows the variance (equivalently standard deviation) to increase in exchange for faster hashing. We set this parameters so that each partition is expected to be within 5% of its mean.

Entropy-Learned Hashing Reduces Partitioning Time. Table 4 presents the speedups of Entropy-Learned Hashing for the three configurations we examine. Entropy-Learned Hashing dramatically improves the hashing computation as can be seen by the left side of Table 4, with increases in speed of above 3 for 4 of the 5 datasets and speedups of up to 14.1 . Partitioning by writing out positional identifiers, seen in the middle column of Table 4, is similar, with increases in speed of greater than 2 for 4 of the 5 datasets and speedups of up to 6.2 . Thus, the results show that the computational cost of partitioning is significantly cheaper using Entropy-Learned Hashing. At the same time, writing out large amounts of data can limit the benefits of using ELH for partitioning, as seen in the right side of Table 4. By writing out long-key strings at each iteration of the partitioning, limitations on write bandwidth limit gains from Entropy-Learned Hashing. Still, even in this case the speedups can be as much as 20%, and additionally CPU usage is reduced which frees up the CPU for other tasks.

	Pure hashing		Pos. id.		Data	
# Par.	64	1024	64	1024	64	1024
UUID	1.44	0.95	1.44	0.95	1.44	0.95
Wp.	0.92	1.02	0.92	1.02	0.93	1.02
Wiki	1.35	1.01	1.35	1.01	1.35	1.01
Hn	2.06	1.00	2.06	1.00	2.05	1.00
Ggle	1.09	1.08	1.09	1.08	1.09	1.08

Table 5: The relative standard deviations of Entropy-Learned Hashing and full-key hashing are similar.

Partitioning quality is maintained using Entropy-Learned Hashing. Table 5 presents normalized relative standard deviation for partitioning, where relative standard deviation is obtained by dividing the standard deviation by the average. We calculate relative standard deviation for both full-key and Entropy-Learned Hashing and normalize the Entropy-Learned Hashing to the full-key hashing. As Table 5 shows, the normalized relative standard deviations concentrate around one, which shows that the partitions produced by the full-key hashing and the partitions produced by the entropy-learned hashing are similar. In the case they are not, such as for Hacker News with 64 partitions, the relative standard deviation of Entropy-Learned Hashing is less than 3% so that partitions are within 3% of their expected number of items on average.

6.6 Large Key Experiments

A key point of Entropy-Learned Hashing is that its runtime is independent of key size. While this already provides speedups for medium sized keys such as URLs and text, this speedup is much larger for large keys such as file blocks. To show the effects of Entropy-Learned Hashing on large keys, we repeat our previous experiments for hash tables, bloom filters, and partitioning but with synthetic random keys of 8192 bytes each. Figure 11 shows the results. For hash tables with all successful lookups, the benefits of Entropy-Learned Hashing are bounded because comparing keys limits the throughput of these tasks. For hash table lookups that are misses, Bloom filter probes, and partitioning, this speedup is unbounded and can be one to two orders of magnitude. Thus, when keys are large, speedups from using Entropy-Learned Hashing can be extremely sizable with respect to runtime.

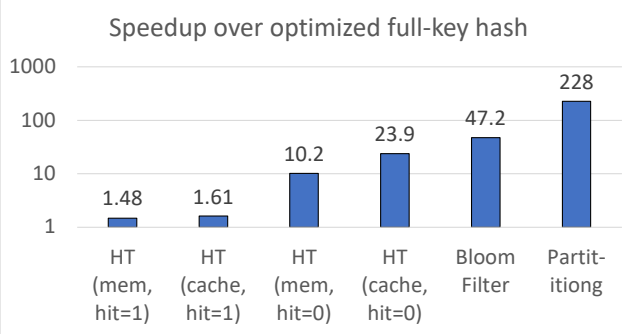

Figure 11: Entropy-Learned Hashing provides orders of magnitude speedups at large key sizes.

6.7 Training Time

Beyond the benefits produced from using Entropy-Learned Hashing in data structures, an additional goal is that the algorithms to select bytes are cheap to perform. To evaluate the training time, we show the runtimes of our training algorithms on the full Google dataset. Table 6 shows the results, displaying algorithm run times for our naive implementation which keeps all data points at each iteration, and for our optimized implementation which discards unique keys after each iteration. There are three main takeaways. First, the training time is reasonable for all sizes of contiguous bytes chosen, with runtimes between several minutes and several seconds. Second, pruning items which are unique from the dataset after each iteration produces substantial runtime benefits (if an item is unique on some subset of bytes, adding new bytes cannot create a collision for that item). Third, as the size of the contiguous byte locations we choose increases, the runtime decreases significantly because there are fewer options at each iteration and because after fewer iterations the number of data items that are non-unique is low (making each step, i.e. Algorithm 2, fast).

# bytes	1	4	8
Optimized	214 s	11.6 s	6.4 s
Naive	29 min.	13 min	5 min

Table 6: Training runtime

6.8 Additional Experiments

The paper focuses on a curated set of experiments which best showcase the properties of Entropy-Learned Hashing. Due to space constraints, this leaves out several experiments which cover other key metrics. Briefly, this includes experiments on 1) the efficiency of creating Entropy-Learned Hash data structures, 2) probing separate chaining hash tables, 3) experiments with dependent accesses (i.e. hash table lookups and Bloom filter lookups which must run one after the other instead of in parallel), 4) additional experiments on Bloom filters showing a range of desired false positive rates, and 5) experiments showing robustness properties. We include all of these results in the technical report [29].

7 RELATED WORK

Entropy & Hashing. Chung, Mitzenmacher, and Vadhan's work [18, 41] explains why current hash functions perform well, hypothesizing that data randomness is the reason this occurs. Our work makes a step forward to change the practice of hashing by recognizing this randomness, choosing how much and which parts of the data we need to hash, and making hash functions cheaper.

Non-Cryptographic Hash Functions. New hash functions are continually designed and fitted to modern processors [64]. This includes works with some form of data-independent randomness guarantees such as multiply-shift [22], CLHash [38], and tabulation hashing[51, 72]. These works are complementary to Entropy-Learned Hashing as they can be modified to work over subsets of bytes to achieve even better speeds.

Data-Dependent hashing. Hash functions which depend on the data have been considered before. For point lookups, this includes perfect hashing [26] and learned hash indexes [35]. Both these methods introduce computational overhead while trying to reduce the number of collisions. Entropy-Learned Hashing is complementary to such works and it can be used in conjunction with these techniques to get both better computation and a lower number of collisions. An additional line of work which is related in terms of learning from data but for a very different application is using data to learn what items are approximately nearest neighbors [65].

Cryptographic Hash Functions. Cryptographic hash functions such as MD5 [60], SHA1 [24] and newer variants have more stringent measures on the ability to invert hash function outputs, but can and have been used for hash-based data structures. A cryptographic hash function specifically designed for hash-based data structures is SipHash [8]. While development of newer cryptographic hash functions has made cryptographic hashing faster, it remains an order of magnitude slower than non-cryptographic hashing [19].

8 CONCLUSION & FUTURE WORK

This paper introduces Entropy-Learned Hashing, a way to reduce the cost of hash functions by modelling the input data to produce hash functions that give just enough randomness. We demonstrate that this approach leads to substantial benefits in terms of computational speed on hash tables, Bloom filters, and load balancing. We also derived key relationships between the entropy of the data source and the performance of data structures, deriving how much entropy is needed for each data structure when given access to a suitably good hash function.

Future work includes investigating the relationship between the distribution of source data and the necessary operations inside the hash function. For instance, experimentally the fastest hash function for integer tables for most datasets is multiply-shift [59]; however, theoretically it is known that certain datasets produce non-constant access times for linear probing when using this hash function [48]. A dataset-specific view of this approach would illuminate when and why we can use this hash function.

9 ACKNOWLEDGEMENTS

This work is partially funded by the USA Department of Energy project DE-SC0020200 and by the Swiss National Science Foundation Early Postdoc Mobility scholarship P2ELP2_199749.

REFERENCES

[1] [n.d.]. gcc libstdc++ hash. https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/libsupc%2B%2B/hash_bytes.cc. Accessed: 2021-05-23.

[2] 2015. Hacker News Posts. https://www.kaggle.com/hacker-news/hacker-news-posts. Accessed: 2021-05-23.

[3] 2019. Linker Throughput Improvement in Visual Studio 2019. https://devblogs.microsoft.com/cppblog/linker-throughput-improvement-in-visual-studio-2019/.

[4] Jayadev Acharya, Alon Orlitsky, Ananda Theertha Suresh, and Himanshu Tyagi. 2017. Estimating Renyi Entropy of Discrete Distributions. *IEEE Transactions on Information Theory* 63, 1 (2017), 38–56. https://doi.org/10.1109/TIT.2016.2620435

[5] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inf. Process. Lett.* 101, 6 (March 2007), 255–261.

[6] Austin Appleby. [n.d.]. murmurhash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3. Accessed: 2021-05-23.

[7] Austin Appleby. [n.d.]. smhasher suite. https://github.com/aappleby/smhasher. Accessed: 2021-05-23.

[8] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT 2012*, Steven Galbraith and Mridul Nandi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–508.

[9] Eric Balkanski, Sharon Qian, and Yaron Singer. 2021. Instance specific approximations for submodular maximization. In *International Conference on Machine Learning*. PMLR, 609–618.

[10] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[11] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[12] Colin R. Blyth. 1980. Expected Absolute Error of the Usual Estimator of the Binomial Parameter. *The American Statistician* 34, 3 (1980), 155–157. http://www.jstor.org/stable/2683873

[13] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. 13, 12 (2020), 2649–2661.

[14] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. 2002. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*. 636–646.

[15] Andrei Z. Broder. 1997. On the resemblance and containment of documents.. In *SEQUENCES*, Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro, and James A. Storer (Eds.). IEEE, 21–29. http://dblp.uni-trier.de/db/conf/sequences/sequences1997.html#Broder97

[16] Nathan Bronson and Xiao Shi. [n.d.]. Open-sourcing F14 for faster, more memory-efficient hash tables. https://engineering.fb.com/2019/04/25/developer-tools/f14/.

[17] J. Lawrence Carter and Mark N. Wegman. 1977. Universal Classes of Hash Functions (Extended Abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 106–112.

[18] Kai-Min Chung, Michael Mitzenmacher, and Salil Vadhan. 2013. Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. *Theory of Computing* 9, 30 (2013), 897–945. https://doi.org/10.4086/toc.2013.v009a030

[19] Yann Collet. [n.d.]. xxHash. https://cyan4973.github.io/xxHash/. Accessed: 2021-05-23.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[21] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3448016.3457273

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

[23] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (April 2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[24] D. Eastlake and P. Jones. 2001. RFC3174: US Secure Hash Algorithm 1 (SHA1).

[25] P. Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* (2007), 137–156.

[26] Michael L. Fredman, Michael L. Fredman, Michael L. Fredman, Michael L. Fredman, Janos Komlos, Janos Komlos, Janos Komlos, Janos Komlos, Endre Szemeredi, Endre Szemeredi, Endre Szemeredi, and Endre Szemeredi. 1982. Storing a sparse table with O(1) worst case access time. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 165–169. https://doi.org/10.1109/SFCS.1982.39

[27] Google. [n.d.]. Abseil Common Libraries. https://github.com/abseil/abseil-cpp.

[28] Jason Gregory. 2009. *Game engine architecture* (1 ed.). Taylor & Francis Ltd.

[29] Brian Hentschel, Utku Sirin, and Stratos Idreos. [n.d.]. Entropy-Learned Hashing Technical Report. https://bhentsch.github.io/doc/EntropyLearnedHashingTechnicalReport.pdf. https://github.com/AnonymousSigmod2022/EntropyLearnedHashing/blob/master/TechnicalReport.pdf

[30] Intel. 2021. Intel VTune Amplifier XE Performance Profiler. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[31] Adam Kirsch and Michael Mitzenmacher. 2006. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms*. Springer, 456–467.

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

[33] Chun-Wa Ko, Jon Lee, and Maurice Queyranne. 1995. An exact algorithm for maximum entropy sampling. *Operations Research* 43, 4 (1995), 684–691.

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

[35] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 489–504.

[36] Matt Kulukundis. [n.d.]. Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step. https://www.youtube.com/watch?v=ncHmEUmJZf4.

[37] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment* 12, 5 (2019), 502–515.

[38] Daniel Lemire and Owen Kaser. 2016. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering* 6, 3 (2016), 171–185.

[39] Linux. 2021. Perf Wiki. https://perf.wiki.kernel.org/.

[40] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 334–350. https://doi.org/10.1145/3341302.3342076

[41] Michael Mitzenmacher and Salil Vadhan. 2008. Why simple hash functions work: Exploiting the entropy in a data stream. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 746–755.

[42] Michael David Mitzenmacher and Alistair Sinclair. 1996. *The Power of Two Choices in Randomized Load Balancing*. Ph.D. Dissertation. AAI9723118.

[43] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. 2016. ntHash: recursive nucleotide hashing. *Bioinformatics* 32, 22 (07 2016), 3492–3494. https://doi.org/10.1093/bioinformatics/btw397 arXiv:https://academic.oup.com/bioinformatics/article-pdf/32/22/3492/19397493/btw397_Sup.pdf

[44] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functions—I. *Mathematical programming* 14, 1 (1978), 265–294.

[45] Hyeonwoo Noh, Andre Araujo, Jack Sim, and Bohyung Han. 2016. Large-Scale Image Retrieval with Attentive Deep Local Features. *International Conference on Computer Vision (ICCV)* (2016). http://arxiv.org/abs/1612.06321

[46] Maciej Obremski and Maciej Skorski. 2017. Renyi Entropy Estimation Revisited. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA (LIPIcs, Vol. 81)*, Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:15.

[47] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385. http://dblp.uni-trier.de/db/journals/acta/acta33.html#ONeilCGO96

[48] Anna Pagh, Rasmus Pagh, and Milan Ružić. 2011. Linear Probing with 5-Wise Independence. *SIAM Rev.* 53, 3 (Aug. 2011), 547–558. https://doi.org/10.1137/110827831

[49] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* 51, 2 (May 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[50] Mihai Pătraşcu and Mikkel Thorup. 2010. On the k-Independence Required by Linear Probing and Minwise Independence. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 715–726.

[51] Mihai Patrascu and Mikkel Thorup. 2011. The Power of Simple Tabulation Hashing. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing* (San Jose, California, USA) *(STOC '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1993636.1993638

[52] W. W. Peterson. 1957. Addressing for Random-Access Storage. *IBM Journal of Research and Development* 1, 2 (1957), 130–146. https://doi.org/10.1147/rd.12.0130

[53] Geoff Pike and Jyrki Alakuijala. 2011. CityHash. https://github.com/google/cityhash.

[54] Geoff Pike and Jyrki Alakuijala. 2014. FarmHash. https://github.com/google/farmhash.

[55] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 755–766. https://doi.org/10.1145/2588555.2610522

[56] M. V. Ramakrishna. 1988. Hashing Practice: Analysis of Hashing and Universal Hashing. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. Association for Computing Machinery, New York, NY, USA, 191–199. https://doi.org/10.1145/50202.50223

[57] M. V. Ramakrishna. 1989. Practical Performance of Bloom Filters and Parallel Free-Text Searching. *Commun. ACM* 32, 10 (Oct. 1989), 1237–1239. https://doi.org/10.1145/67933.67941

[58] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., USA.

[59] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 96–107. https://doi.org/10.14778/2850583.2850585

[60] R. Rivest. 1992. RFC1321: The MD5 Message-Digest Algorithm.

[61] Kenneth A. Ross. 2007. Efficient Hash Probes on Modern Processors. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 1297–1301.

[62] Utku Sirin and Anastasia Ailamaki. 2020. Micro-Architectural Analysis of OLAP: Limitations and Opportunities. *Proc. VLDB Endow.* 13, 6 (2020), 840–853.

[63] Oracle ZFS Steve Tunstall. 2017. DeDupe 2.0. https://blogs.oracle.com/wonders-of-zfs-storage/dedupe-20-v2. Accessed: 2021-05-23.

[64] Reini Urban. [n.d.]. SMHasher - Reini Urban Fork. https://github.com/rurban/smhasher. Accessed: 2021-05-23.

[65] Jingdong Wang, Ting Zhang, jingkuan song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (2018), 769–790. https://doi.org/10.1109/TPAMI.2017.2699960

[66] Yi Wang, Diego Barrios Romero, Daniel Lemire, and Li Jin. 2020. Modern Non-Cryptographic Hash Function and Pseudorandom Generator. (2020).

[67] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-Core Radix Sort. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II* (Bordeaux, France) *(Euro-Par'11)*. Springer-Verlag, 160–169.

[68] Mark N. Wegman and J.Lawrence Carter. 1981. New hash functions and their use in authentication and set equality. *J. Comput. System Sci.* 22, 3 (1981), 265–279. https://doi.org/10.1016/0022-0000(81)90033-7

[69] Oracle ZFS. 2019. ZFS Deduplication. https://blogs.oracle.com/bonwick/zfs-deduplication-v2. Accessed: 2021-05-23.

[70] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. 2020. SAHA: A String Adaptive Hash Table for Analytical Databases. *Applied Sciences* 10, 6 (2020). https://doi.org/10.3390/app10061915

[71] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)* (Virtual Event, China) *(DAMON'21)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3465998.3466002

[72] A. Zobrist. 1990. A New Hashing Method with Application for Game Playing. *ICGA Journal* 13 (1990), 69–73.