Appendix: Entropy Learned Hashing
10x Faster Hashing with Controllable Uniformity

ANONYMOUS AUTHOR(S)

This document accompanies Entropy-Learned Hashing and covers aspects of the paper which were not covered in the main body of the paper. This covers proofs regarding Entropy-Learned Hashing and linear probing, discussion of robustness for Entropy-Learned Hashing data structures, and additional experiments.

Appendix A Proofs

A.1 Notation & Problem Setup

The notation and problem setup are mostly the same as in the original paper. The only difference is that linear probing in this document resolves collisions by going to the left, i.e. checking $i, i-1, i-2, \ldots$ rather than $i, i+1, \ldots$. This is to be consistent with Knuth's proof for linear probing given in [3]. A reminder of the rest of the notation can be seen in the table below.

A.2 Linear Probing

As in the main document, we start our proof by going over full-key hashing, then anayze partial-key hashing with a fixed dataset, and finally cover partial-kay hashing with random data. Before starting the proof, we briefly cover the results.

A.2.1 Results The expected number of comparisons for querying a missing key and average number of comparisons for an existing key when using full-key hashing are:

$$\mathbb{E}[P'] \leq \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \tag{1}$$

$$\mathbb{E}[P] \leq \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \tag{2}$$

Letting $c = \sum_x z_x^2$ be the number of collisions and $d = \sum_{x:z_x \geq 2} z_x$ the number of duplicated keys, the costs for partial-key hashing are

$$\mathbb{E}[P'] \leq \begin{cases} \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2}\right) & \text{if } z_y = 0 \\ \frac{z_y}{1-\alpha} + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2} & \text{if } z_y > 0 \end{cases} \tag{3}$$

$$\begin{aligned} \mathbb{E}[P] &\leq \frac{n-d}{2n} + \frac{1}{2}Q_0(m,n) + \frac{c}{m}Q_0(m,n) + \frac{c+d}{2n}Q_0(m,d) \\ &\approx \left(\frac{1}{2} + \frac{c}{n}\right)\left(1 + \frac{1}{1-\alpha}\right) \end{aligned} \tag{4}$$

For random data from an i.i.d. source with Renyi entropy H_2, these translate to the following bounds:

$$\mathbb{E}[P'] \leq \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) + n2^{-H_2(L(X))}\frac{3}{2(1-\alpha)^2}$$

$$\mathbb{E}[P] \leq \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) + n2^{-H_2(L(X))}\left(1 + \frac{1}{1-\alpha}\right)$$

Notation	Definition
X, x	keys stored in the filter or hash table
H, h	hash function filter or hash table
Y, y	query key in filter or hash table
m	size of filter (in bits) or table (in slots)
n	number of keys in filter or table
K	set of keys
$S_{\|L}$	multi-set of keys conditioned on L. Equal to $(K_{\|L}, z)$ where $K_{\|L}$ consists of all partial keys, and z maps each key $x \in K_{\|L}$ to the number of times it appears in K. z_x will be used as shorthand for $z(x)$ throughout.
α	fill of hash table: $\frac{n}{m}$
P'	number of comparisons to find non-existing key
P	average # of comparisons to retrieve an existing key

Table 1. Notation used throughout the paper

Comparing Partial-Key Hashing with Full-Key Hashing. The above equations, when fully analyzed, make a rather intuitive point: when there are only a few duplicates in K_L so that it closely approximates full-key hashing, the number of comparisons required by full-key and partial-key hashing are close. The general form of each equation for partial key hashing is that it is the same as full-key hashing plus a small penalty term. These penalty terms go to 0 as the keys in partial-key hashing become more distinct.

For all equations, we note the equations above are relatively tight bounds as long as α is not close to 1. This is reasonable to assume as all hash tables keep $\alpha < 0.95$ in practice because of the extremely bad behavior of linear probing tables as $\alpha \to 1$, wherein they degrade to linear scans. Most hash table implementations keep α between 0.25 and 0.85, with higher alpha leading to better memory usage but slower query times.

A.2.2 Analysis of Full Key Linear Probing To prove (3) and (4), we start by proving (1) and (2), first proven by Donald Knuth. Our proof initially follows steps taken in The Art of Computer Programming[3], however we deviate from the proof given there because that approach does not generalize to partial-key hashing. Additionally, we believe this proof is simpler to follow than the proof in [3].

Counting Hash Sequences. For n distinct items, we have m^n possible ways to assign them to $[m] = \{0, \ldots, m-1\}$, all of which are equally likely by our hashing assumptions. Of all possible hash sequences, we first consider how many leave position 0 empty in the hash table.

Theorem 1. *The number of hash sequences such that n items are hashed into* $[1, m-1]$*, and no overflow occurs so that position 0 is empty is*

$$\begin{cases} (1 - \frac{n}{m}) \cdot h(m, n) & \text{if } 0 \le n \le m \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $h(m, n)$ *is the number of ways to hash n objects into* $[0, m-1]$*.*

This is a slight reformulation of the way it is stated in [3] so that it generalizes to partial-key hashing. To prove the theorem, note that hash sequences which hash n items into $[1, m-1]$ and do not overflow into position 0 are precisely the same as those that leave position 0 empty when hashing into a hash table of size m and resolving collisions via linear probing. This probability is $1 - \frac{n}{m}$ by the circular symmetry of linear probing. Thus, for full-key hashing there are $(1 - \frac{n}{m})m^n$ sequences which leave location 0 empty.

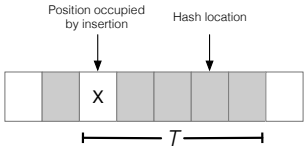
The next theorem gives the probability that a run of occupied slots begins at location 1 in the table and stops at location k.

Theorem 2. *Let $g(m,n,k)$ be the number of sequences when hashing into a hash table of size m that leave position 0 empty, positions 1 to $k-1$ filled, and k empty. This is*

$$g(m,n,k) = \binom{n}{k-1}\frac{1}{k}\frac{m-n-1}{m-k}h(k,k-1)h(m-k,n-k+1) \tag{6}$$

This theorem simply builds off of Theorem 1 as there are $\binom{n}{k-1}$ ways to choose the $k-1$ elements in the run starting at 1, and then by Theorem 1 there are $\frac{1}{k}h(k,k-1)$ and $\frac{m-n-1}{m-k}h(m-k,n-k-1)$ ways to create the two subsequences with locations $0,k$ empty for each choice of the $k-1$ elements.

Analyzing Chain Length. We use the two theorems above to analyze a random variable T which represents the length counting from the empty position which a new item is inserted into to the final occupied position before the next empty position (going from left to right). The figure below shows an example.



If we know the expected value for T for a new item to be inserted, we can get its average distance from its hash location by the uniform randomness of hashing. Namely, for any chain of length T, each location in the chain is equally likely as an initial hash location, so $\mathbb{E}[P'|T] = \frac{1}{T}\sum_{i=1}^{T} i = \frac{1}{2} + \frac{1}{2}T$. It follows that

$$\mathbb{E}[P'] = \mathbb{E}[\mathbb{E}[P'|T]] = \frac{1}{2} + \frac{1}{2}\mathbb{E}[T]$$

To see how the two theorem above apply to our analysis of T, assume that the insertion key hashes to location a. If the insertion key is hashed into a chain of length t, then for some $t \geq 1, 0 \leq k \leq t-1$, we have $a-k$ empty, $a-k+1,\ldots,a+(t-k-1)$ full, and $a+(t-k)$ empty. We note by symmetry that the same number of sequences produce a chain of this nature as which produce a chain such that 0 is empty, $1,\ldots,t-1$ is full, t is empty. Since we have t choices for k, it then follows that there are $t \cdot g(m,n,t)$ hash sequences for which the insertion item is inserted into a chain of length t.

The approach taken in Knuth's art of computer programming is to analyze $E[T]$ directly by using Abel's Binomial Theorem, i.e.

$$(x+y)^n = \sum_k \binom{n}{k} x(x-kz)^{k-1}(y+kz)^{n-k}$$

to find a recursive relationship in the formula for $\mathbb{E}[T]$. While correct, this approach feels magical as the use of Abel's Binomial is unintuitive (at least to the authors). Additionally, it does not generalize to the case of partial-key hashing, and so we take a different approach.

Analyzing Chain Length By Connecting Chain Length and the Probability of Chain Inclusion. Our approach to analyze $\mathbb{E}[T]$ is to look at the probability that each item is in the same probe chain as the inserted item. Namely, if we let $x_1,\ldots,x_n$ be the keys inserted into the hash table, then we have that

$$\mathbb{E}[T] = 1 + \sum_{i=j}^{n} P(x_j \in T)$$

Here we slightly abuse notation to have T both represent the length of the chain on the left and the set of objects in the chain on the right. For $P(x_j \in T)$, we connect this probability to the expected chain length conditioned on that $x_j \in T$.

More concretely, let T_i be the chain length of an inserted item if i of the n items in the hash table have the same hash value as the item to be inserted. All other items are distributed uniformly at random and independently of the hash value of the to be inserted item. The following theorem holds.

Theorem 3. *If C represents a set of i values which are fixed to have the same hash location as the newly inserted item, then for $x \notin C$, we have*

$$P(x \in T_i) = \frac{1}{m}\,\mathbb{E}[T_{i+1}]$$

Proof. We first derive the formula for $\mathbb{E}[T_i]$. Assume as before that the insertion key hashes to location a. If the new item is hashed into a chain of length t, then for some $t \geq 1, 0 \leq k \leq t-1$, we have $a-k$ empty, $a-k+1, \ldots, a+(t-k-1)$ full, and $a+(t-k)$ empty. Again by symmetry, the number of sequences that produce a chain of this nature is the same as the number of sequences that produce a chain such that 0 is empty, $1, \ldots, t-1$ is full, t is empty, and the i items hash to location k. Let this number be $g(m,n,t,i,k)$. Then $f(m,n,t,i) = \sum_{k\geq 0} g(m,n,t,i,k)$ is the total number of sequences such that the new item is hashed into a chain of length t.

The value for $f(m,n,t,i)$ is calculable directly as this is identical to hashing $t-i-1$ items of multiplicity 1 and 1 value of multiplicity i into the first t slots keeping slot 0 empty, and hashing $n-(t-1)$ items into the final $m-t$ slots. Using theorem 1, and setting $n' = n-i$, we have

$$\begin{aligned} f(m,n,t,c_y) &= t \cdot g(m,n',t-i) \\ &= \binom{n'}{t-i-1} t^{t-i-1}(m-t)^{n-t}(m-n-1) \end{aligned}$$

The expected value of T_i is then $m^{-n'} \sum_{t\geq 1} t \cdot f(m,n,t,c_y)$.

We now calculate the probability that $x \notin C$ is in T_i. Given that i items match the location of the newly inserted item, the number of chains which contain $x \notin C$ is

$$\begin{aligned} &= \sum_t \binom{n'-1}{t-i-2} t^{t-i-2}(m-t)^{n-t}(m-n-1) \\ &= \sum_t t f(m,n,t,i+1) \end{aligned}$$

where $n' = n-i$. Dividing by the number of hash sequences, $m^{n'} = m^{n'-1}m$ gives that $P(x \in T_i) = \frac{1}{m}\,\mathbb{E}[T_{i+1}]$. □

Finishing the Proof. Using these relationship between $\mathbb{E}[T_i]$ and $P(x \in T_i)$, we have

$$\begin{aligned} E[T] &= 1 + \sum_{i=1}^{n} P(x_i \in T_0) \\ &= 1 + \frac{n}{m}\,\mathbb{E}[T_1] \\ &= 1 + \frac{n}{m}(2 + \frac{n-1}{m}\,\mathbb{E}[T_2])) \end{aligned}$$

where here we use that $\mathbb{E}[T_i] = (i+1) + \sum_{x \notin C} P(x \in T_i)$. If we continue expanding the values of $\mathbb{E}[T_i]$, we get that

$$\begin{aligned}\mathbb{E}[T] &= 1 + 2\frac{n}{m} + 3\frac{n^{\underline{2}}}{m^2} + \cdots + \frac{n!}{m^n} \\ &= Q_1(m, n)\end{aligned}$$

where

$$Q_r(m, n) = \sum_{k \geq 0} \binom{k+r}{k} \frac{n^{\underline{k}}}{m^k}$$

It follows that

$$\mathbb{E}[P'] = \frac{1}{2} + \frac{1}{2} Q_1(m, n)$$

recovering the value proven in [3]. Noting that $\frac{n^{\underline{k}}}{m^k} \leq \alpha^k$ and using that $\sum_{k \geq 0}(k+1)\alpha^k = d/d\alpha(\sum_{k \geq 0} \alpha^k) = (1-\alpha)^{-2}$ gives the desired bound $\mathbb{E}[P'] \leq \frac{1}{2}(1 + (1-\alpha)^{-2})$

Average Cost to Query an existing item. To go from $\mathbb{E}[P']$, the cost for a newly inserted key, to $\mathbb{E}[P]$, the average cost to query a key in the table, we average over the cost to insert the first n keys. This is because the cost to find a key in linear probing is the same as the cost to insert that key. So

$$\mathbb{E}[P] = \frac{1}{n} \sum_{i \geq 0}^{n} \mathbb{E}[P'_i]$$

where $\mathbb{E}[P'_i]$ is the expected cost to insert a key if there are i items in the table. Thus

$$\begin{aligned}\mathbb{E}[P] &= \frac{1}{2} + \frac{1}{2n} \sum_{i=0}^{n-1} Q_1(m, i) \\ &= \frac{1}{2} + \frac{1}{2n} \sum_{i=0}^{n-1} \sum_{k \geq 0} (k+1) \frac{i^{\underline{k}}}{m^k} \\ &= \frac{1}{2} + \frac{1}{2n} \sum_{k \geq 0} \sum_{i=0}^{n-1} (k+1) \frac{i^{\underline{k}}}{m^k}\end{aligned}$$

Now we use the rules of "finite calculus", which says that if $\Delta f(x) = f(x+1) - f(x) = g(x)$, then $\sum_{i=a}^{b} g(x) = f(b+1) - f(a)$. Here we have $\Delta x^{\underline{k}} = k x^{\underline{k-1}}$, so $\sum_{x=0}^{n-1} k x^{\underline{k-1}} = n^{\underline{k}}$. Plugging this in above, we have

$$\begin{aligned}\mathbb{E}[P] &= \frac{1}{2} + \frac{1}{2n} \sum_{k \geq 0} \sum_{i=0}^{n-1} (k+1) \frac{i^{\underline{k}}}{m^k} \\ &= \frac{1}{2} + \frac{1}{2n} \sum_{k \geq 0} \frac{n^{\underline{k+1}}}{m^k} \\ &= \frac{1}{2} + \frac{1}{2} Q_0(m, n-1)\end{aligned}$$

A.2.3 Proof of Costs Using Partial-Key Hashing Let C be the set of multiplicities for partial-key hashing. Throughout, we use the notation for a multiset C that $|C| = |K_L|$ is the number of distinct partial-key elements and $||C|| = \sum_{x \in C} z(x_1)$ is the total number of elements in C.

Theorem 1,2 and 3 Restatements. Looking at theorem 1, it's proof holds for partial-key hashing. Namely, if $n \leq m$ objects with multiplicities C are hashed into locations $[1, m-1]$, then the number of hash sequences such that 0 is empty is

$$(1 - \frac{n}{m})h(m, C)$$

where $h(m, C)$ is the number of ways to hash the objects into $[0, m-1]$. For a given multiset C, we have $h(m, C) = m^{|C|}$.

To generalize theorem 2, we start by analyzing the form of equation (6) for $g(m, n, k)$. The equation consists of three parts: 1) ways to divide the set of n objects into one set with $k-1$ items and another with $n-k+1$ items, 2) ways to hash the $k-1$ and $n-k+1$ items into their arrays of size k,$m-k$, and 3) the scaling factors that say only $\frac{1}{k}, \frac{m-n-1}{m-k}$ of those leave slots 0 and k empty. It follows that

$$g(m, C, k) = \sum_{\substack{C_1 \subset C \\ ||C_1|| = k}} \frac{1}{k} \frac{m-n-1}{m-k} h(k, C_1) h(m-k, C \setminus C_1)$$

Theorem 3 also holds for multisets, but we need new notation to be clearer about what items are included in the new chain. We define $T_{C'}$ to be the expected chain length of a new item if all items in the multiset $C' \subseteq C$ are guaranteed to have the same hash value as the newly inserted item. If we define $f(m, C, t, C')$ to be the probability that this chain is of length t, then $f(m, C, t, C') = tg(m, C \setminus C', t-i)$ analogously to before.

The number of chains in the sample space for $T_{C'}$ which contain $x \notin C'$ is $\sum_{t \geq 1} tf(m, C, t, C'')$ where $C'' = C' \cup \{x\}$, and so $P(x \in T_{C'}) = \frac{1}{m} \mathbb{E}[T_{C''}]$.

Notational Shorthand. To make the equations in the subsequent sections easier to read, we extend the shorthand notation $z_x = z(x)$ to sets, i.e. $z_{C'} = \sum_{x \in C'} z_x = ||C'||$. Additionally, we define $C_{2+} = x : z_x \geq 2$, so that it represents the set of keys which are not unique in C.

Bounds on $\mathbb{E}[T_{C'}]$. We prove the following bound by induction on the set of unallocated objects, i.e. $C \setminus C'$:

$$\mathbb{E}[T_{C'}] \leq z_{C'} Q_0(m, n - z_{C'}) + Q_1(m, n - z_{C'}) + \sum_{x \in (C \setminus C')_{2+}} \frac{z_x^2}{m} Q_1(m, n - z_{C'})$$

Base case: Our base case is all sets such that $C' = C$, i.e. all objects in C are guaranteed to hash to the same location as the newly inserted item. The length of T is guaranteed to be $z_{C'} + 1 \leq z_{C'} Q_0(m, 0) + Q_1(m, 0)$.

Induction Steps: Assume our induction hypothesis holds for all sets $C_1, C_1' \subset C_1$ with $|C_1 \setminus C_1'| \leq k$. Assume that the set C, C' we are calculating our expectation over is such that $|C \setminus C'| = k+1$. As shorthand, let $C^- = C \setminus C'$

$$\begin{aligned}
\mathbb{E}[T_{C'}] &\leq (1 + z_{C'}) + \sum_{x \in C^-} P(x \in T_{C'}) \\
&= (1 + z_{C'}) + \frac{1}{m} \sum_{x \in C^-} z_x \Big[(z_x + z_{C'}) Q_0(m, n - z_{C'} - z_x) + Q_1(m, n - z_{c'} - z_x) + \sum_{y \in C^-, y \neq x} \frac{z_y^2}{m} Q_1(m, n - z_{C'} - z_x) \Big] \\
&\leq (1 + z_{C'}) + \frac{n - z_{C'}}{m} z_{C'} Q_0(m, n - z_{C'} - 1) + \sum_{x \in C^-} \frac{z_x^2 + z_x}{m} Q_0(m, n - z_{C'} - 1) \\
&\quad + \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1) + \sum_{x \in C^-} z_x^2 \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1) \\
&= z_{C'} (1 + \frac{n - z_{C'}}{m} Q_0(m, n - z_{C'} - 1)) && (7) \\
&\quad + 1 + \frac{n - z_{C'}}{m} Q_0(m, n - z_c - 1) + \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1) && (8) \\
&\quad + \sum_{x \in C^-_{2+}} \frac{z_x^2}{m} (Q_0(m, n - z_c - 1) + \frac{n - z_{C'}}{m} Q_1(m, n - z_c - 1)) && (9)
\end{aligned}$$

Now, using the rules $\frac{n}{m} Q_1(m, n-1) = Q_1(n, m) - Q_0(n, m)$ and $\frac{n}{m} Q_0(m, n-1) = Q_0(m, n) - 1$ we can reduce equations (10),(11), and (12) respectively to $z_{C'} Q_0(m, n - z_{C'})$, $Q_1(m, n - z_{C'})$, and $\sum_{x \in (C \setminus C')_{2+}} \frac{z_x^2}{m} Q_1(m, n - z_{C'})$ respectively. This gives the required result on $\mathbb{E}[T_{C'}]$.

Connecting $\mathbb{E}[T]$ to $\mathbb{E}[P']$. Depending on whether $z_y = 0$, the connection between chain length and probe length changes. For items which match no other partial-keys in the dataset, we can use the same uniformity assumptions as before to recover that $\mathbb{E}[P] = \frac{1}{2} + \frac{1}{2}\mathbb{E}[T]$. For items which match at least one partial-key, this uniformity assumption does not hold, and indeed items with larger c_y values are more likely to be hashed towards the beginning of their chain. Thus, we use that $P \leq T$ to say $\mathbb{E}[P] \leq \mathbb{E}[T]$, and note this is a loose bound when z_y is small. The result, overall, is equation 3, restated here for convenience.

$$\mathbb{E}[P'] \leq \begin{cases} \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2}\right) & \text{if } z_y = 0 \\ \frac{z_y}{1-\alpha} + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2} & \text{if } z_y > 0 \end{cases}$$

Average query cost for an existing key. The average cost to query a key in linear probing is unaffected by the order in which keys are inserted. Thus, we may assume any insertion order we desire in order to ease the analysis of the average number of probes for existing keys.

Because our analysis in the prior section created looser bounds for $z_y > 0$, namely in that there was no uniformity assumption on where items were in a chain, we add all duplicate keys first and then all non-duplicate

keys after. Expanding this out, we have

$$\begin{aligned}
\mathbb{E}[P] &\leq \frac{1}{n}\Bigg(\sum_{x\in C_{2+}}\frac{z_x^2}{2}Q_0(m,d)+\sum_{i=0}^{d-1}\Big[Q_1(m,i)+\sum_{x\in C_{2+}}\frac{z_x^2}{m}Q_1(m,i)\Big]\Bigg)+\frac{1}{2n}\Big[\sum_{i=d}^{n-1}\big(1+Q_1(m,i)+\sum_{x\in C_{2+}}\frac{z_x^2}{m}Q_1(m,i)\big)\Big]\\
&\leq \frac{n-d}{n}+\frac{1}{2}Q_0(m,n-1)+\frac{c+d}{2n}Q_0(m,d-1)+\frac{c}{2m}(Q_0(m,n-1)+\frac{d}{n}Q_0(m,d-1))\\
&\approx \frac{1}{2}(1+\frac{1}{1-\alpha})+\frac{c}{n}+\frac{c}{m}\frac{1}{1-\alpha}\\
&\leq (\frac{1}{2}+\frac{c}{n})(1+\frac{1}{1-\alpha})
\end{aligned}$$

Here we use that $\alpha_d = \frac{d}{m} \approx 0$ so that $\frac{1}{1-\alpha_d} \approx 1$. Because we are interested in the case that duplicate items are rare, this is a very good approximation.

Random Data. To go from fixed data to random data, we condition via Adam's law. For querying for a missing key, we first rewrite the expectation as

$$\mathbb{E}[P'] \leq \frac{1}{2}+\mathbb{1}_{z_y\neq 0}(\frac{z_y}{1-\alpha}-\frac{1}{2})+(\frac{1}{2}+\frac{1}{2}\mathbb{1}_{z_y\neq 0})\frac{1}{(1-\alpha)^2}+(\frac{1}{2}+\frac{1}{2}\mathbb{1}_{z_y\neq 0})\sum_x\frac{z_2^2}{m}\frac{1}{(1-\alpha)^2}$$

In this form, we then use $\mathbb{E}[z_y] \leq n2^{-H_2(L(X))}$, which gives

$$\begin{aligned}
\mathbb{E}[P'] &\leq \frac{1}{2}(1+\frac{1}{(1-\alpha)^2})+\frac{n2^{-H_2(L(X))}}{1-\alpha}+\frac{n2^{-H_2L(X))}}{2(1-\alpha)^2}+\sum_x\frac{\alpha n2^{-H_2(L(X))}}{(1-\alpha)^2}\\
&\leq \frac{1}{2}(1+\frac{1}{(1-\alpha)^2})+n2^{-H_2(L(X))}\frac{3}{2(1-\alpha)^2}
\end{aligned}$$

More straightforwardly,

$$\mathbb{E}[P] \leq \frac{1}{2}(1+\frac{1}{1-\alpha})+n2^{-H_2(L(X))}(1+\frac{1}{1-\alpha})$$

This concludes the analysis of linear probing, proving the initial equations given in the paper.

Appendix B Robustness of Entropy-Learned Hashing Data Structures

Robustness. For most algorithms and data structures, including those listed here, there is a trade-off between their robustness and their expected performance. How to choose between the expected performance and robustness is highly dependent on the application; certain applications might accept a 20% better average performance for a 1% chance at 2X worse performance whereas others may not. For Entropy-Learned Hashing and other techniques which exist as a base layer of potential use to many applications, the goal is to make this trade-off minimal, so that large gains in expected performance come at little cost for robustness. For Entropy-Learned Hashing, the amount of robustness for the hashing task depends on 1) the assumptions of Entropy-Learned Hashing, and 2) the task at hand. We start by discussing the assumptions of Entropy-Learned Hashing and then discuss Entropy-Learned Hashing for hash tables, Bloom Filters, and partitioning individually.

Assumptions of Entropy-Learned Hashing. Entropy-Learned Hashing makes weak assumptions, namely that data which are somewhat random remain somewhat random. Thus, unlike the usual case for learning where shifts in data distributions almost always cause degradations in performance, shifts in data distribution for Entropy-Learned Hashing cause no change in performance as long as data stays random. In particular, if a distribution D_1 shifts to distribution D_2 this only causes a performance shift if $H_2(L(D_2)) << H_2(L(D_1))$. Thus the main robustness of Entropy-Learned Hashing comes from the fact that changes in distribution are unlikely to

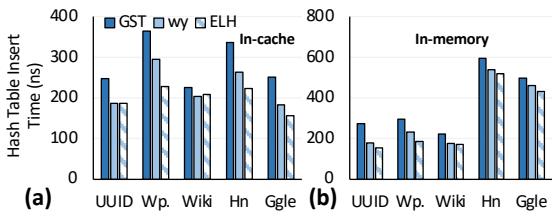


Fig. 1. Entropy-Learned Hashing reduces insert times for hash tables across datasets, and data sizes.

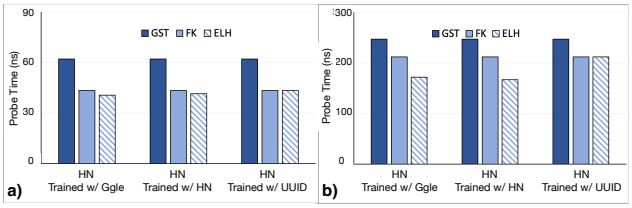


Fig. 2. Entropy-Learned Hashing reduces insert times for hash tables across datasets, and data sizes.

degrade performance. Still, we must address the case where D_2 actually becomes predictable on the bytes we choose to hash. We now cover this for each data structure.

Hash Tables. Entropy-Learned Hashing for hash tables is the most robust. This is for multiple reasons. First, if collisions are as expected on keys in the dataset, queries for both keys in the data and not in the data return quickly (see equations (4) and (5) for example). Second, we can monitor collisions during insertions with little overhead by counting the displacement from the initial hash position. Thus, this together with reason 1 means we have guaranteed good performance at all times. Third, Entropy-Learned Hashing can rehash if collisions ever deviate what is expected. The simplest way to do this is to revert to a full-key hash function, although other approaches such as backup bytes to include are an interesting future direction. Thus performance for hash tables can be guaranteed to be at least as good as traditional hashing (at the expense of code complexity) while usually being substantially better.

Bloom Filters. Bloom filters are less robust than hash tables. Their first defense in terms of robustness is the weak assumptions, namely that different distributions work for Entropy-Learned Hashing as long as the distribution is still random. Their second defense is that we can check that the data matches the desired distribution, namely because the # of set bits concentrates sharply around their expected value [1]. We use this fact to estimate the number of unique items in the filter. If data items are not as expected, or if queries are substantially different than the inserted items leading to a larger FPR, the filter must be rebuilt.

Partitioning. 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. This is simple to implement and does not affect in-memory tasks like radix-partitioning much. For more complex tasks for which partitioning happens across a more expensive medium such as a network, approaches such as using the least loaded of d-bins would be of use [2].

Appendix C Additional Experiments

This Section covers the additional experiments described in Section 6.6. This includes experiments on 1) the efficiency of creating Entropy-Learned Hash data structures, 2) probing separate chaining hash tables, 3) experiments showing robustness properties, 4) experiments with dependent accesses (i.e. hash table lookups and Bloom filter lookups which must run one after the other instead of in parallel), and 5) additional experiments on Bloom filters showing different false positive rates.

1. Entropy-Learned Hashing Reduces Hash Table Insert Time. Entropy-Learned Hashing works similarly for inserts as it works for probes with hit rate = 1. Figure 1 presents hash table insert times for the five real-world datasets we examine. The figure shows that Entropy-Learned hashing provides a 1.16× to 1.3× speedup over its closest competitor thanks to its reduced hash computation both for in-cache and in-memory data sizes. Thus, the speedups seen in query performance on Entropy-Learned hash tables carry over to performance on the insertion of data.

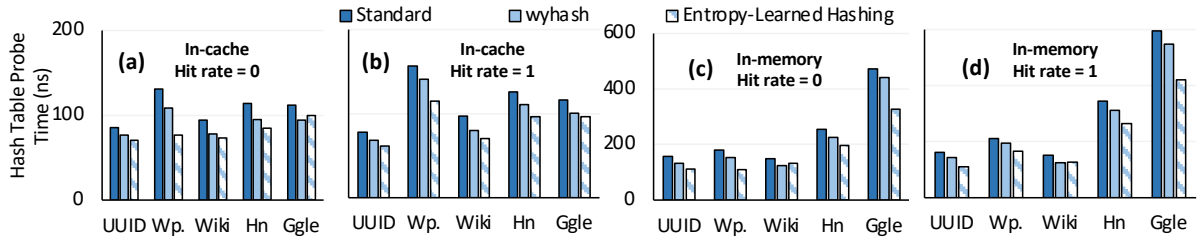


Fig. 3. Entropy-Learned Hashing reduces the probe time of standard chaining hash tables.

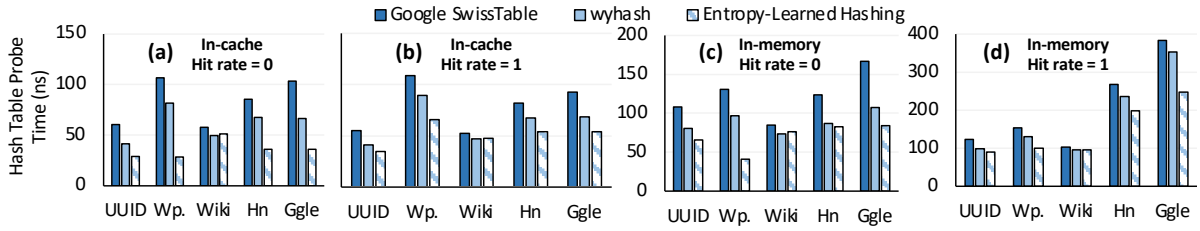


Fig. 4. Entropy-Learned Hashing reduces dependent probe times for hash tables across datasets, data sizes, and hit rates.

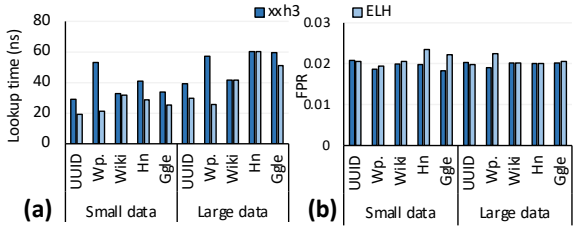


Fig. 5. Improving Bloom filter lookup time (a) and false positive rates (b) for small and large data sizes for dependent lookups.

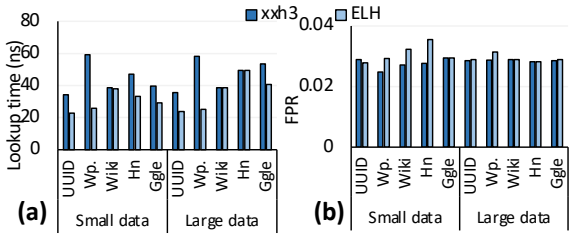


Fig. 6. Improving regular Bloom filter lookup time (a) and false positive rates (b) for small and large data sizes.

2. Separate Chaining Experiments. To show that Entropy-Learned Hashing works with separate chaining hash maps as well as linear probing hash maps, we integrate Entropy-Learned Hashing into std::unordered_map. The results can be seen in Figure 3. The key takeaway is that, as expected, Entropy-Learned Hashing reduces the probe times across datasets, data sizes, and hit rates, with this improvement being up to 1.72×. Thus Entropy-Learned Hashing works for separate chaining tables in addition to linear probing tables. The improvement is slightly lower than what is seen for SwissTable; this is mainly because std::unordered_map is much slower as a baseline and so portions of the hash table probe that are not the hash function take up more of the computation. A second change is that std::unordered_map is only a tiny amount faster when querying for non-existent keys than when querying for existent keys. This is again because unordered_map lacks optimizations present in SwissTable. Thus, a second takeaway of Figure 3 is that Entropy-Learned Hashing provides larger speedups for more optimized hash tables.

3. Robustness of Entropy-Learned Hashing. To test the robustness of Entropy-Learned hash tables, we test the performance of them when their training sets, i.e the set of sampled past queries and data items, does not match their test distribution of actual inserted and queried items. To test this, we vary the distribution given as a data sample and then insert and query items from the Hacker News dataset. Figures 2a,b show the results when using the large data size and querying for missing items (Fig. 2a) and existing items (Fig. 2b). As expected, when the Hacker News dataset is used to gather data as well as to insert and query items, we see speedups of 5% to 30%. More importantly, as Figures 2a and 2b show, when using Google URLs as a dataset to analyze ahead of time, using the Hacker News dataset to insert and query items still produces speedups of 5% to 27%. This shows the case where the test data is different in distribution but still random enough on the bytes of choice; in this case Entropy-Learned Hashing still provides speedups over full-key hashing. When using the UUID dataset as a training dataset, the user ids from UUID and posted urls from Hacker News are very different and Entropy-Learned Hashing defaults to using the full-key hash function as described in Section 2 of the technical report (and Section 5 of the main paper). Thus, in this case it provides no speedups but does not degrade performance.

4. Entropy-Learned Hashing Reduces Dependent Probe Times. Entropy-Learned Hashing reduces probe times for data structures when the probes are dependent (i.e. they cannot be batched). Figure 4 presents hash table probe times, and shows Entropy-Learned Hashing provides 1.18x to 2.9x speedups across different datasets, data sizes and hit rates. Figure 5 shows similar results for dependent Bloom filter lookups, where every lookup requires the result of the preceding lookup to start. The figure shows that Entropy-Learned Hashing significantly reduces the filter lookup times and provides a 1.16× to 2.5× speedup across datasets and data sizes.

The speedups are consistently lower for dependent lookups than they are for independent lookups. The reason is that independent lookups benefit from inter-lookup parallelism, as we discussed in Section 6.3. However, we observe a significant speedup even for dependent lookups. To illustrate, Entropy-Learned Hashing is 1.42× faster than wyhash for hash table probes on the Google dataset with large data and high hit rate. This is because there are two types of memory-level parallelism that Entropy-Learned Hashing benefits from: (i) inter-lookup parallelism, and (ii) intra-lookup parallelism. While independent hash table probes benefit from both inter- and intra-lookup parallelism, dependent hash table probes benefit from only intra-lookup parallelism. As a result, Entropy-Learned Hashing improves hash table probe time more when the probes are independent than it improves when the probes are dependent. To verify our hypothesis, we examined the MLP value for Entropy-Learned Hashing and wyhash for dependent hash table probes and found the MLP value is 2.0 for Entropy-Learned Hashing compared to 1.6 for wyhash. Thus ELH provides both parallelism benefits and computational benefits and produces consistent speedups for dependent lookups across small and large data sizes.

5. Experiments with different FPRs for Bloom Filters. The experiments in the main body of the text focused on throughput optimized filters, in particular using register-blocked Bloom filters from [4]. Figure 6 shows the performance of Entropy-Learned Hashing on traditional Bloom filters at a lower false positive rate of 1%. The results are extremely similar to Section 6.4, and so we don't cover them in detail. Entropy-Learned Hashing again provides benefits in lookup performance of up to 2.4× at negligible (and tunable) increases to the false positive rate.

References

[1] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[2] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 318–326, New York, NY, USA, 1992. Association for Computing Machinery.

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

[4] H. Lang, T. Neumann, A. Kemper, and P. Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment*, 12(5):502–515, 2019.