The Power of Simple Tabulation Hashing

Mihai Pătraşcu
AT&T Labs

Mikkel Thorup
AT&T Labs

February 9, 2011

Abstract

Randomized algorithms are often enjoyed for their simplicity, but the hash functions used to yield the desired theoretical guarantees are often neither simple nor practical. Here we show that the simplest possible tabulation hashing provides unexpectedly strong guarantees.

The scheme itself dates back to Carter and Wegman (STOC'77). Keys are viewed as consisting of c characters. We initialize c tables $T_1, \ldots, T_c$ mapping characters to random hash codes. A key $x = (x_1, \ldots, x_q)$ is hashed to $T_1[x_1] \oplus \cdots \oplus T_c[x_c]$, where $\oplus$ denotes xor.

While this scheme is not even 4-independent, we show that it provides many of the guarantees that are normally obtained via higher independence, e.g., Chernoff-type concentration, min-wise hashing for estimating set intersection, and cuckoo hashing.

Contents

1 Introduction

An important target of the analysis of algorithms is to determine whether there exist *practical* schemes, which enjoy mathematical *guarantees* on performance.

Hashing and hash tables are one of the most common inner loops in real-world computation, and are even built-in "unit cost" operations in high level programming languages that offer associative arrays. Often, these inner loops dominate the overall computation time. Knuth gave birth to the analysis of algorithms in 1963 [**?**] when he analyzed linear probing, the most popular practical implementation of hash tables. Assuming a perfectly random hash function, he bounded the expected number of probes. However, we do not have perfectly random hash functions. The approach of algorithms analysis is to understand when simple and practical hash functions work well. The most popular multiplication-based hashing schemes maintain the $O(1)$ running times when the sequence of operations has sufficient randomness [**?**]. However, they fail badly even for very simple input structures like an interval of consecutive keys [**?**, **?**, **?**], giving linear probing an undeserved reputation of being non-robust.

On the other hand, the approach of algorithm design (which may still have a strong element of analysis) is to construct (more complicated) hash functions providing the desired mathematical properties. This is usually done in the influential k-independence paradigm of Wegman and Carter [**?**]. It is known that 5-independence is sufficient [**?**] and necessary [**?**] for linear probing. Then one can use the best available implementation of 5-independent hash functions, the tabulation-based method of [**?**, **?**].

Here we analyze simple tabulation hashing. This schemes views a key x as a vector of c characters $x_1, \ldots, x_c$. For each character position, we initialize a totally random table T_i, and then use the hash function

$$h(x) = T_1[x_1] \oplus \cdots \oplus T_c[x_c].$$

This is a well-known scheme dating back at least to Wegman and Carter [**?**]. From a practical view-point, tables T_i can be small enough to fit in fast cache, and the function is probably the easiest to implement beyond the bare multiplication. However, the scheme is only 3-independent, and was therefore assumed to have weak mathematical properties.

The challenge in analyzing simple tabulation is the significant dependence between keys. Nevertheless, we show that the scheme works in some of the most important randomized algorithms, including linear probing and several instances when $\Omega(\lg n)$-independence was previously needed. We confirm our findings by experiments: simple tabulation is competitive with just one 64-bit multiplication, and the hidden constants in the analysis appear to be very acceptable in practice.

In many cases, our analysis gives the first provably good *implementation* of an algorithm which matches the algorithm's conceptual simplicity if one ignores hashing.

Problems. Consider the following statements about simple and popular randomized algorithms:

- The worst-case query time of chaining is $O(\lg n / \lg \lg n)$ with high probability (w.h.p.). More generally, when distributing balls into bins, the bin load obeys Chernoff bounds.
- Linear probing runs in expected $O(1)$ time per operation.
- Cuckoo hashing: Given two tables of size $m \geq (1+\varepsilon)n$, it is possible to place a ball in one of two randomly chosen locations without *any* collision, with probability $1 - O(\frac{1}{n})$.
- Given two sets A, B, we have $\Pr_h[\min h(A) = \min h(B)] = \frac{|A \cap B|}{|A \cup B|}$. This can be used to quickly estimate the intersection of two sets, and follows from a property called *minwise independence*: for any $x \notin S$, $\Pr_h[x < \min h(S)] = \frac{1}{|S|+1}$.

As defined by Wegman and Carter [**?**] in 1977, a family $\mathcal{H} = \{h : [u] \to [m]\}$ of hash functions is k-independent if for any distinct $x_1, \ldots, x_k \in [u]$, the hash codes $h(x_1), \ldots, h(x_k)$ are independent random variables, and the hash code of any fixed x is uniformly distributed in $[m]$.

Chernoff bounds continue to work with high enough independence [**?**]; for instance, independence $\Theta(\frac{\lg n}{\lg \lg n})$ suffices for the bound on the maximum bin load. For linear probing, 5-independence is sufficient [**?**] and necessary [**?**]. For cuckoo hashing, $O(\lg n)$-independence suffices and at least 6-independence is needed [**?**]. While minwise independence cannot be achieved, one can achieve ε-minwise independence with the guarantee $(\forall) x \notin S, \Pr_h[x < \min h(S)] = \frac{1 \pm \varepsilon}{|S|+1}$. For this, $\Theta(\lg \frac{1}{\varepsilon})$ independence is sufficient [**?**] and necessary [**?**]. (Note that the ε is a lower bound on how well set intersection can be approximated, with any number of independent experiments.)

The canonical construction of k-independent hash functions is a random degree $k-1$ polynomial in a prime field, which has small representation but $\Theta(k)$ evaluation time. This is unappealing when $k \approx \lg n$. Siegel [**?**] shows that a family with superconstant independence but $O(1)$ evaluation time requires $\Omega(u^\varepsilon)$ space, i.e. it requires tabulation. He also gives a solution that uses $O(u^{1/c})$ space, $c^{O(c)}$ evaluation time, and achieves $u^{\Omega(1/c^2)}$ independence (which is superlogarithmic, at least asymptotically).

Competitive implementations of polynomial hashing simulate arithmetic modulo Mersenne primes via bitwise operations. Even so, tabulation-based hashing with $O(u^{1/c})$ space and $O(ck)$ evaluation time is significantly faster [**?**]. While polynomial hashing may perform better than its independence suggests, we have no positive example yet. On the tabulation front, we have one example of a good hash function that is not formally k-independent: cuckoo hashing works with an ad hoc hash function that combines space $O(n^{1/c})$ and polynomials of degree $O(c)$ [**?**].

1.1 Our results

Here we provide an analysis of simple tabulation showing that it has many of the desirable properties above. For most of our applications, we want to rule out certain obstructions with high probability. This follows immediately if certain events are independent, and the algorithms design approach is to pick a hash function guaranteeing this independence, usually in terms of a highly independent hash function.

Instead we here stick with simple tabulation with all its dependencies. This means that we have to struggle in each individual application to show that the dependencies are not fatal. However, from an implementation perspective, this is very attractive, leaving us with one simple and fast schemes for (almost) all our needs.

In all our results, we assume the number of characters is $c = O(1)$. The constants in our bounds will depend on c. Our results use a rather diverse set of techniques analyzing the table dependencies in different types of problems. For chaining and linear probing, we rely on some concentration results, which will also be used as a starting point for the analysis of min-wise hashing. Theoretically, the most interesting part is the analysis for cuckoo hashing, with a very intricate study of the interaction between the two hash functions.

Chernoff bounds. We first show that simple tabulation preserves Chernoff-type concentration:

Theorem 1. *Consider hashing n balls into m bins by simple tabulation. Let q be an additional* query ball*, and define X_q as the number of regular balls that hash into a bin chosen as a function of $h(q)$. Let $\mu = \mathbf{E}[X_q] = \frac{n}{m}$. Then for any constant γ:*

$$\Pr[X_q \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\Omega(\mu)} + n^{-\gamma} \qquad \Pr[X_q \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\Omega(\mu)} + n^{-\gamma}$$

Let us contrast this to the standard Chernoff bounds [**?**]:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \qquad \Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} \tag{1}$$

As opposed to the standard bounds, our result can only achieve polynomially small probability, i.e. at least $n^{-\gamma}$ for any desired constant γ. In addition the exponential dependence on μ is reduced by a constant, which depends on the choice of γ and c.

An alternative way to understand the bound is that our tail bound depends exponentially on $\varepsilon\mu$, where ε decays to subconstant as we move further out in the tail. Thus, our bounds are sufficient for any polynomially high probability guarantee. However, compared to the standard Chernoff bound, we would have to tolerate a constant factor more balls in a bin to get the same failure probability.

From Theorem 1, we can derive some standard approximations. For $\delta \in [0,1]$, we get a normal distribution type bound with high probability:

$$\Pr[|X-\mu| > \delta\mu] < e^{-\Omega(\delta^2\mu)} + n^{-\gamma} \tag{2}$$

Also, if $x \geq 2\mu$, we get a Poisson distribution type bound with high probability:

$$\Pr[X > x] < (\mu/x)^{\Omega(x)} + n^{-\gamma} \tag{3}$$

By the union bound (2) implies that no bin receives more than $O(\lg n/\lg\lg n)$ balls w.h.p. This is the first realistic hash function to achieve this fundamental property. Similarly, for linear probing with fill bounded below 1, (3) shows that the longest filled interval is of length $O(\log n)$ w.h.p.

Linear probing. Building on the above concentration bounds, we show that if the table size is $m = (1+\varepsilon)n$, then the expected time per operation is $O(1/\varepsilon^2)$, which asymptotically matches the bound of Knuth [**?**] for a truly random function. In particular, this compares positively with the $O(1/\varepsilon^{13/6})$ bound of [**?**] for 5-independent hashing. We can also show that the variance is constant for constant ε, whereas with 5-independent hashing it was only known to be $O(\log n)$ [**?**, **?**].

Our proof is a new combinatorial reduction that relates the performance of linear probing to concentration bounds. The results hold for any hash function with concentration similar to Theorem 1.

Minwise independence. We show that simple tabulation is ε-minwise independent, for a vanishingly small ε (inversely polynomial in the set size). This would require $\Theta(\log n)$ independence by standard techniques.

Theorem 2. *Consider a set S of $n = |S|$ keys and $q \notin S$. Then with h implemented by simple tabulation:*

$$\Pr[h(q) < \min h(S)] = \frac{1 \pm \varepsilon}{n}, \qquad \textit{where } \varepsilon = O\left(\frac{\lg^2 n}{n^{1/c}}\right).$$

This can be used to estimate the size of set intersection by estimating:

$$\Pr[\min h(A) = \min h(B)] = \sum_{x \in A \cap B} \Pr[x < \min h(A \cup B \setminus \{x\})] = \frac{|A \cap B|}{|A \cup B|} \cdot \left(1 \pm \widetilde{O}\left(\frac{1}{|A \cup B|^{1/c}}\right)\right).$$

Cuckoo hashing. In general, the cuckoo hashing algorithm fails iff the random bipartite graph induced by two hash functions contains a component with more vertices than edges. With truly random hashing, this happens with probability $\Theta(\frac{1}{n})$. Here we study the random graphs induced by simple tabulation, and obtain a rather unintuitive result: the optimal failure probability is inversely proportional to the *cube root* of the set size.

Theorem 3. *Any set of n keys can be placed in two table of size $m = (1+\varepsilon)$ by cuckoo hashing and simple tabulation with probability $1 - O(n^{-1/3})$. There exist sets on which the failure probability is $\Omega(n^{-1/3})$.*

Thus, cuckoo hashing and simple tabulation are an excellent construction for a static dictionary. The dictionary can be built in $O(n)$ time w.h.p., and later every query runs in constant worst-case time with two probes. We note that even though cuckoo hashing requires two independent hash functions, these essentially come for the cost of one in simple tabulation: the pair of hash codes can be stored consecutively, in the same cache line.

However, the lower bound suggests this dictionary is not a good alternative for dynamic dictionaries, where we will have a complete rehash after every $O(n^{4/3})$ operations, in expectation.

Our proof involves a complex understanding of the intricate, yet not fatal dependencies in simple tabulation. The proof is a (complicated) algorithm that assumes that cuckoo hashing has failed, and uses this knowledge to compress the random tables $T_1, \ldots, T_c$ below the entropy lower bound.

Using our techniques it is also possible to show that if n balls are placed in $O(n)$ bins in an online fashion, choosing the least loaded bin at each time, the maximum load is $O(\lg \lg n)$ in expectation.

Fourth moment bounds. An alternative to Chernoff bounds in proving good concentration is to use bounded moments. We analyze the 4th moment of a bin's size, when balls are placed into bins by simple tabulation. For a fixed bin, we show that the 4th moment comes extremely close to that achieved by truly random hashing: it deviates by a factor of $1 + O(4^c/m)$, which is tiny except for a very large number of characters c. This would require 4-independence by standard arguments. This limited 4th moment for a given bin was discovered independently by [**?**].

If we have a designated query ball q, and we are interested in the size of a bin chosen as a function of $h(q)$, the 4th moment of simple tabulation is within a constant factor of that achieved by truly random hashing (on close inspection of the proof, that constant is at most 2). This would require 5-independence by standard techniques. (See [**?**] for a proof that 4-independence can fail quite badly when we want to bound the size of the bin in which q lands.) Our proof exploits an intriguing phenomenon that we identify in simple tabulation: in any fixed set of 5 keys, one of them has a hash code that is independent of the other four's hash codes.

Unlike our Chernoff-type bounds, the constants in the 4th moment bounds can be analyzed quite easily, and are rather tame. Compelling applications of 4th moment bounds were given by [**?**] and [**?**]. In [**?**], it was shown that any hash function with a good 4th moment bound suffices for a nonrecursive version of quicksort, routing on the hypercube, etc. In [**?**], linear probing is shown to have constant expected performance if the hash function is a composition of universal hashing down to a domain of size $O(n)$, with a strong enough hash function on this small domain (i.e. any hash function with a good 4th moment bound).

Pseudorandom numbers. The tables used in simple tabulation should be small to fit in the first level of cache. Thus, filling them with truly random numbers would not be difficult (e.g. in our experiments we use atmospheric noise from `random.org`). If the amount of randomness needs to be reduced further, we

remark that all proofs continue to hold if the tables are filled by a $\Theta(\lg n)$-independent hash function (e.g. a polynomial with random coefficients).

With this modification, simple tabulation naturally lends itself to an implementation of a very efficient pseudorandom number generator. We can think of a pseudorandom generator as a hash function on range $[n]$, with the promise that each $h(i)$ is evaluated once, in the order of increasing i. To use simple tabulation, we break the universe into two, very lopsided characters: $[\frac{n}{R}] \times [R]$, for R chosen to be $\Theta(\lg n)$. Here the second coordinate is least significant, that is, (x, y) represents $xR+y$. During initialization, we fill $T_2[1\,..\,R]$ with R truly random numbers. The values of $T_1[1\,..\,n/R]$ are generated on the fly, by a polynomial of degree $\Theta(\lg n)$, whose coefficients were chosen randomly during initialization. Whenever we start a new row of the matrix, we can spend a relatively large amount of time to evaluate a polynomial to generate the next value r_1 which we store in a register. For the next R calls, we run sequentially through T_2, xoring each value with r_1 to provide a new pseudorandom number. With T_2 fitting in fast memory and scanned sequentially, this will be much faster than a single multiplication, and with R large, the amortized cost of generating r_1 is insignificant. The pseudorandom generator has all the interesting properties discussed above, including Chernoff-type concentration, minwise independence, and random graph properties.

Experimental evaluation. In the appendix, we perform an experimental evaluation of simple tabulation. Our implementation uses tables of 256 entries (i.e. using $c = 4$ characters for 32-bit data and $c = 8$ characters with 64-bit data). The time to evaluate the hash function turns out to be competitive with multiplication-based 2-independent functions, and significantly better than for stronger hash functions.

We then evaluate simple tabulation in applications, in an effort to verify that the constants hidden in our analysis are not too large. We try sets chosen from a dense interval, which is known to be a bad input instance for multiplicative hash functions, as well as a hypercube, which uses the least amount of randomness in simple tabulation (and is intuitively the worst case for this scheme). Simple tabulation turns out to be very robust, both for linear probing and for cuckoo hashing.

Notation. We now introduce some notation that will be used throughout the proofs. We want to construct hash functions $h : [u] \to [m]$. We use simple tabulation with an alphabet of Σ and $c = O(1)$ characters. Thus, $u = \Sigma^c$ and $h(x_1, \ldots, x_c) = \bigoplus_{i=1}^{c} T_i[x_i]$. It is convenient to think of each hash code $T_i[x_i]$ as a fraction in $[0, 1)$ with large enough precision. We always assume m is a power of two, so an m-bit hash code is obtained by keeping only the most significant $\log_2 m$ bits in such a fraction. We always assume the table stores long enough hash codes, i.e. at least $\log_2 m$ bits.

Let $S \subset \Sigma^c$ be a set of $|S| = n$ keys, and let q be a query. We typically assume $q \notin S$, since the case $q \in S$ only involves trivial adjustments (for instance, when looking at the load of the bin $h(q)$, we have to add one when $q \in S$). Let $\pi(S, i)$ be the projection of S on the i-th coordinate, $\pi(S, i) = \{x_i \mid (\forall)x \in S\}$.

We define a *position-character* to be an element of $[c] \times \Sigma$. Then, the alphabets on each coordinate can be assumed to be disjoint: the first coordinate has alphabet $\{1\} \times \Sigma$, the second has alphabet $\{2\} \times \Sigma$, etc. Under this view, we can treat a key x as a *set* of q position-characters (on distinct positions). Furthermore, we can assume h is defined on position characters: $h((i, \alpha)) = T_i[\alpha]$. This definition is extended to keys (sets of position-characters) in the natural way $h(x) = \bigoplus_{\alpha \in x} h(\alpha)$.

When we say with high probability in r, we mean $1 - r^a$ for any desired constant a. Since $c = O(1)$, high probability in $|\Sigma|$ is also high probability in u. If we just say high probability, it is in n.

2 Concentration Bounds

If n elements are hashed into $n^{1+\varepsilon}$ bins by a truly random hash function, the maximum load of any bin is $O(1)$ with high probability. In §2.1, we begin by showing that simple tabulation preserves this guarantee. Building on this, §2.2 shows that the load of any fixed bin obeys Chernoff bounds. In §2.3, we show that this holds even for a bin chosen as a function of the query hash code, $h(q)$.

2.1 Hashing into Many Bins

The notion of peeling lies at the heart of most work in tabulation hashing. If a key from a set of keys contains one position-character that doesn't appear in the rest of the set, its hash code will be independent of the rest. Then, it can be "peeled" from the set, as its behavior matches that with truly random hashing. More formally, we say a set T of keys is peelable if we can arrange the keys of T in some order, such that each key contains a position-character that doesn't appear among the previous keys in the order.

Lemma 4. *Suppose we hash $n \le m^{1-\varepsilon}$ keys into m bins, for some constant $\varepsilon > 0$. For any constant γ, there exists a constant $d = d(\varepsilon, c, \gamma)$, such that: with probability $\ge 1 - m^{-\gamma}$, all bins get less than d keys.*

Proof. We will show that no bin contains more than $d = \min\{(\frac{1+\gamma}{\varepsilon})^c, 2^{(1+\gamma)/\varepsilon}\}$ elements.

We will show that among any d elements, one can find a *peelable* subset of size $t \ge \max\{d^{1/c}, \lg d\}$. Then, a necessary condition for the maximum load of a bin to be at least d is that some bin contain t peelable elements. There are at most $\binom{n}{t} < n^t$ such sets. Since the hash codes of a peelable set are independent, the probability that a fixed set lands into a common bin is $1/m^{t-1}$. Thus, an upper bound on the probability that the maximum load is d can be obtained: $n^t/m^{t-1} = m^{(1-\varepsilon)t}/m^{t-1} = m^{1-\varepsilon t}$. To obtain failure probability $m^{-\gamma}$, set $t = (1+\gamma)/\varepsilon$.

It remains to show that any set T of $|T| = d$ keys contains a large peelable subset. Since $T \subset \pi(T,1) \times \cdots \times \pi(T,c)$, it follows that there exists $i \in [c]$ with $|\pi(T,i)| \ge d^{1/c}$. Pick some element from T for every character value in $\pi(S,i)$; this is a peelable set of $t = d^{1/c}$ elements.

To prove $t \ge \log_2 d$, we proceed iteratively. Consider the coordinate giving the largest projection, $j = \arg\max_i |\pi(T,i)|$. As long as $|T| \ge 2$, $|\pi(T,j)| \ge 2$. Let α be the most popular value in T for the j-th character, and let $T^\star$ contain only elements with α on the j-th coordinate. We have $|T^\star| \ge |T|/|\pi(T,j)|$. In the peelable subset, we keep one element for every value in $\pi(T,j) \setminus \{\alpha\}$, and then recurse in $T^\star$ to obtain more elements. In each recursion step, we obtain $k \ge 1$ elements, at the cost of decreasing $\log_2 |T|$ by $\log_2(k+1)$. Thus, we obtain at least $\log_2 d$ elements overall. □

2.2 Chernoff Bounds for a Fixed Bin

We study the number of keys ending up in a prespecified bin B. The analysis will define a total ording $\prec$ on the space of position-characters, $[c] \times \Sigma$. Then we will analyze the random process by fixing hash values of position-characters $h(\alpha)$ in the order $\prec$. The hash value of a key $x \in S$ becomes known when the position-character $\max_\prec x$ is fixed. For $\alpha \in [c] \times \Sigma$, we define the *group* $G_\alpha = \{x \in S \mid \alpha = \max_\prec x\}$, the set of keys for whom α is the last position-character to be fixed.

The intuition is that the contribution of each group G_α to the bin B is a random variable independent of the previous G_β's, since the elements G_α are shifted by a new hash code $h(\alpha)$. Thus, if we can bound the contribution of G_α by a constant, we can apply Chernoff bounds.

Lemma 5. *There is an ordering $\prec$ such that the maximal group size is $\max_\alpha |G_\alpha| \le n^{1-1/c}$.*

Proof. We start with S being the set of all keys, and reduce S iteratively, by picking a position-character α as next in the order, and removing keys G_α from S. At each point in time, we pick the position-character α that would minimize $|G_\alpha|$. Note that, if we pick some α as next in the order, G_α will be the set of keys $x \in S$ which contain α and contain no other character that hasn't been fixed: $(\forall)\beta \in x \setminus \{\alpha\}, \beta \prec \alpha$.

We have to prove is that, as long as $S \neq \emptyset$, there exists α with $|G_\alpha| \leq |S|^{1-1/c}$. If some position i has $|\pi(S,i)| > |S|^{1/c}$, there must be some character α on position i which appears in less than $|S|^{1-1/c}$ keys; thus $|G_\alpha| \leq S^{1-1/c}$. Otherwise, $\pi(S,i) \leq |S|^{1/c}$ for all i. Then if we pick an arbitrary character α on some position i, have $|G_\alpha| \leq \prod_{j \neq i} |\pi(S,j)| \leq (|S|^{1/c})^{c-1} = |S|^{1-1/c}$. □

From now on assume the ordering $\prec$ has been fixed as in the lemma. This ordering partitions S into at most n non-empty groups, each containing at most $n^{1-1/c}$ keys. We say a group G_α is *d-bounded* if no bin contains more than d keys from G_α.

Lemma 6. *Assume the number of bins is $m \geq n^{1-1/(2c)}$. For any constant γ, there exists a constant d_γ such that, with probability $\geq 1 - n^\gamma$, all groups are d-bounded.*

Proof. Since $|G_\alpha| \leq n^{1-1/c} \leq m^{1-1/(2c)}$, w.h.p. there are at most $O(1)$ keys from G_α in any bin, by Lemma 4. The conclusion follows by union bound over the $\leq n$ groups. □

Chernoff bounds (see [**?**, Theorem 4.1]) consider independent random variables $X_1, X_2, \cdots \in [0,d]$. Let $X = \sum_i X_i$, $\mu = \mathbf{E}[X]$, and $\delta > 0$, the bounds are:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu/d} \qquad \Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/d} \tag{4}$$

Let X_α be the number of elements from G_α landing in the bin B. We are quite close to applying Chernoff bounds to the sequence X_α, which would imply the desired concentration around $\mu = \frac{n}{m}$. Two technical problems remain: X_α's are not d-bounded in the worst case, and they are not independent.

To address the first problem, we define the sequence of random variables $\hat{X}_\alpha$ as follows: if G_α is d-bounded, let $\hat{X}_\alpha = X_\alpha$; otherwise $\hat{X}_\alpha = |G_\alpha|/m$ is a constant. Observe that $\sum_\alpha \hat{X}_\alpha$ coincides with $\sum_\alpha X_\alpha$ if all groups are d-bounded, which happens with probability $1 - n^{-\gamma}$. Thus a probabilistic bound on $\sum_\alpha \hat{X}_\alpha$ is a bound on $\sum_\alpha X_\alpha$ up to an additive $n^{-\gamma}$ in the probability.

Finally, the $\hat{X}_\alpha$ variables are not independent: earlier position-character dictate how keys cluster in a later group. Fortunately, the proof of the Chernoff bounds from [**?**] holds even if the distribution of each X_i is a function of $X_1, \ldots, X_{i-1}$, as long as $\mathbf{E}[X_i \mid X_1, \ldots, X_{i-1}]$ is a constant. We claim that this is the case: regardless of the hash codes for $\beta < \alpha$, $\mathbf{E}[\hat{X}_\alpha] = |G_\alpha|/m$.

Observe that whether or not G_α is d-bounded is known immediately before $h(\alpha)$ is fixed in the order $\prec$. Indeed, α is the last position-character to be fixed for any key in G_α, so the hash codes of all keys in G_α have been fixed up to an xor with $h(\alpha)$. This final shift by $h(\alpha)$ is common to all the keys, so it cannot change the whether or not two elements landed in the same bin. After fixing all hash codes $\beta \prec \alpha$, we decide whether $\hat{X}_\alpha = X_\alpha$ or $\hat{X}_\alpha$ is a constant. In the former case, note that $h(\alpha)$ remains a uniform random variable, so the expected number of elements in B remains $|G_\alpha|/m$.

This shows that the number of keys in bin B obeys Chernoff bounds as in Theorem 1.

2.3 The Load of a Query-Dependent Bin

When we are dealing with a special key q (a query), we may be interested in the load of a bin B_q, chosen as a function of the query's hash code, $h(q)$. We show that the above analysis also works for the size of B_q, up to small constants. The critical change is to insist that the query position-characters come first in our ordering $\prec$:

Lemma 7. *There is an ordering $\prec$ placing the characters of q first, in which the maximal group size is $2 \cdot n^{1-1/c}$.*

Proof. After placing the characters of q at the beginning of the order, we use the same iterative construction as in Lemma 5. Each time we select the position-character α minimizing $|G_\alpha|$, place α next in the order $\prec$, and remove G_α from S. It suffices to prove that, as long as $S \neq \emptyset$, there exists a position-character $\alpha \notin q$ with $|G_\alpha| \leq 2 \cdot |S|^{1-1/c}$. Suppose in some position i, $|\pi(S,i)| > |S|^{1/c}$. Even if we exclude the query character q_i, there must be some character α on position i that appears in at most $|S|/(|\pi(S,i)|-1)$ keys. Since $S \neq \emptyset$, $|S|^{1/c} > 1$, so $|\pi(S,i)| \geq 2$. This means $|\pi(S,i)| - 1 \geq |S|^{1/c}/2$, so α appears in at most $2|S|^{1-1/c}$ keys. Otherwise, we have $\pi(S,i) \leq |S|^{1/c}$ for all i. Then, for any character α on position i, we have $|G_\alpha| \leq \prod_{j \neq i} |\pi(S,j)| \leq |S|^{1-1/c}$. □

The lemma guarantees that the first nonempty group contains the query alone, and all later groups have random shifts that are independent of the query hash code. We lost a factor two on the group size, which has no effect on our asymptotic analysis. In particular, all groups are d-bounded w.h.p. Letting X_α be the contribution of G_α to bin B_q, we see that the distribution of X_α is determined by the hash codes fixed previously (including the hash code of q, fixing the choice of the bin B_q). But $\mathbf{E}[X_\alpha] = |G_\alpha|/m$ holds irrespective of the previous choices. Thus, Chernoff bounds continue to apply to the size of B_q.

3 Minwise Independence

We will prove that:

$$\frac{1}{n} \cdot \left(1 - \frac{O(\lg n)}{n^{1/c}}\right) \;\leq\; \Pr[h(q) < \min h(X)] \;\leq\; \frac{1}{n} \cdot \left(1 + \frac{O(\lg^2 n)}{n^{1/c}}\right) \tag{5}$$

The lower bound is relatively simple, and is shown in §3.1. The upper bound is significantly more involved and appears in §3.2.

For the sake of the analysis, we divide the output range $[0,1)$ into $\frac{n}{\ell}$ bins, where $\ell = \gamma \lg n$ for a large enough constant γ. Of particular interest is the minimum bin $[0, \frac{\ell}{n})$. We choose γ sufficiently large for the Chernoff bounds of Theorem 1 to guarantee that the minimum bin in non-empty w.h.p.: $\Pr[\min h(X) < \frac{\ell}{n}] \geq 1 - \frac{1}{n^2}$.

In §3.1 and §3.2, we assume that hash values $h(x)$ are binary fractions of infinite precision (hence, we can ignore collisions). It is easy to see that (5) continues to hold when the hash codes have $(1 + \frac{1}{c}) \lg n$ bits, even if ties are resolved adversarially. Let $\tilde{h}$ be a truncation to $(1 + \frac{1}{c}) \lg n$ bits of the infinite-precision h. We only have a distinction between the two functions if q is the minimum and $(\exists) x \in S : \tilde{h}(x) = \tilde{h}(q)$. The probability of a distinction is bounded from above by:

$$\Pr\left[\tilde{h}(q) \leq \tfrac{\ell}{n} \;\wedge\; (\exists) x \in S : \tilde{h}(x) = \tilde{h}(q)\right] \;\leq\; \tfrac{\ell}{n} \cdot \left(n \cdot \tfrac{1}{n^{1+1/c}}\right) \;\leq\; \tfrac{O(\lg n)}{n^{1+1/c}}$$

We used 2-independence to conclude that $\{h(q) < \frac{\ell}{n}\}$ and $\{\tilde{h}(x) = \tilde{h}(q)\}$ are independent.

Both the lower and upper bounds start by expressing:

$$\Pr[h(q) < \min h(S)] = \int_0^1 f(p)\mathrm{d}p, \qquad \text{where } f(p) = \Pr[p < \min h(S) \mid h(q) = p].$$

For truly random hash functions, $\Pr[p < \min h(S) \mid h(q) = p] = (1-p)^n$, since each element has an independent probability of $1-p$ of landing about p.

3.1 Lower bound

For a lower bound, it suffices to look at the case when q lands in the minimum bin:

$$\Pr[h(q) < \min h(S)] \geq \int_0^{\ell/n} f(p)\mathrm{d}p, \qquad \text{where } f(p) = \Pr[p < \min h(S) \mid h(q) = p]$$

We will now aim to understand $f(p)$ for $p \in [0, \frac{\ell}{n}]$. In the analysis, we will fix the hash codes of various position-characters in the order $\prec$ given by Lemma 7. Let $h(\prec\alpha)$ done the choice for all position-characters $\beta \prec \alpha$.

Remember that $\prec$ starts by fixing the characters of q first, so: $q_1 \prec \cdots \prec q_c \prec \alpha_0 \prec \alpha_1 \prec \cdots$ Start by fixing $h(q_1), \ldots, h(q_c)$ subject to $h(q) = x$.

When it is time to fix some position-character α, the hash code of any key $x \in G_\alpha$ is a constant depending on $h(\prec\alpha)$ **xor** the random quantity $h(\alpha)$. This final xor makes $h(x)$ uniform in $[0,1)$. Thus, for any choice of $h(\prec\alpha)$, $\Pr[h(z) < p \mid h(\prec\alpha)] = p$. By the union bound, $\Pr[p < \min h(G_\alpha) \mid h(\prec\alpha)] \geq 1 - p \cdot |G_\alpha|$. This implies that:

$$f(p) \;=\; \Pr[p < \min h(S) \mid h(q) = p] \;\geq\; \prod_{\alpha \succ q_c} (1 - p \cdot |G_\alpha|). \tag{6}$$

To bound this product from below, we use the following lemma:

Lemma 8. *Let $p \in [0,1]$ and $k \geq 0$, where $p \cdot k \leq \sqrt{2} - 1$. Then $1 - p \cdot k > (1-p)^{(1+pk)k}$.*

Proof. First we note a simple proof for the weaker statement $(1-pk) < (1-p)^{\lceil (1+pk)k \rceil}$. However, it will be crucial for our later application of the lemma that we can avoid the ceiling.

Consider t Bernoulli trials, each with success probability p. The probability of no failures occurring is $(1-p)^t$. By the inclusion-exclusion principle, applied to the second level, this is bounded from above by:

$$(1-p)^t \;\leq\; 1 - t \cdot p + \binom{t}{2} p^2 \;<\; 1 - (1 - \tfrac{pt}{2}) t \cdot p$$

Thus, $1 - kp$ can be bounded from below by the probability that no failure occurs amount t Bernoulli trials with success probability p, for t satisfying $t \cdot (1 - \frac{pt}{2}) \geq k$. This holds for $t \geq (1 + kp)k$.

We have just shown $1 - p \cdot k > (1-p)^{\lceil (1+pk)k \rceil}$. Removing the ceiling requires an "inclusion-exclusion" inequality with a non-integral number of experiments t. Such an inequality was shown by Gerber [**?**]: $(1-p)^t \leq 1 - \alpha t + (\alpha t)^2/2$, even for fractional t. Setting $t = (1+pk)k$, our result is a corollary of Gerber's inequality:

$$\begin{aligned}(1-p)^t \;&\leq\; 1 - pt + \tfrac{(pt)^2}{2} \;=\; 1 - p(1+pk)k + \tfrac{1}{2}(p(1+pk)k)^2 \\ &=\; 1 - pk - (1 - \tfrac{(1+pk)^2}{2})(pk)^2 \;\leq\; 1 - pk. \quad \square\end{aligned}$$

The lemma applies in our setting, since $p < \frac{\ell}{n} = O(\frac{\lg n}{n})$ and all groups are bounded $|G_\alpha| \le 2 \cdot n^{1-1/c}$. Note that $p \cdot |G_\alpha| \le \frac{\ell}{n} \cdot 2n^{1-1/c} = O(\ell/n^{1/c})$. Plugging into (6):

$$f(p) \;\ge\; \prod_{\alpha \succ q_c} (1 - p \cdot |G_\alpha|) \;\ge\; \prod_{\alpha \succ q_c} (1-p)^{|G_\alpha|(1+\ell/n^{1/c})} \;\ge\; (1-p)^{n \cdot (1+\ell/n^{1/c})}.$$

Let $m = n \cdot (1 + \ell/n^{1/c})$. The final result follows by integration over p:

$$\begin{aligned}
\Pr[h(q) < \min h(S)] \;&\ge\; \int_0^{\ell/n} f(p)\mathrm{d}p \ge \int_0^{\ell/n} (1-p)^m \mathrm{d}p \\
&=\; \frac{-(1-p)^{m+1}}{m+1}\Big|_{p=0}^{\ell/n} \;=\; \frac{1-(1-\ell/n)^{m+1}}{m+1} \\
&>\; \frac{1-e^{-\ell}}{m+1} \;>\; \frac{1-1/n}{n(1+\ell/n^{1/c})} \;=\; \frac{1}{n} \cdot \left(1 - \frac{O(\lg n)}{n^{1/c}}\right)
\end{aligned}$$

3.2 Upper bound

As in the lower bound, it will suffice to look at the case when q lands in the minimum bin:

$$\Pr[h(q) < h(S)] \;\le\; \Pr[\min h(S) \ge \tfrac{\ell}{n}] + \Pr[h(q) < h(S) \wedge h(q) < \tfrac{\ell}{n}] \;\le\; \tfrac{1}{n^2} + \int_0^{\ell/n} f(p)\mathrm{d}p$$

To bound $f(p)$, we will fix position-characters in the order $\prec$ from Lemma 7, subject to $h(q) = p$. In the lower bound, we could analyze the choice of $h(\alpha)$ even for the worst-case choice of $h(\prec\alpha)$. Indeed, no matter how the keys in G_α arranged themselves, when shifted randomly by $h(\alpha)$, they failed to land below p with probability $1 - p|G_\alpha| \ge (1-p)^{(1+o(1))|G_\alpha|}$.

For an upper bound, we need to prove that keys from G_α do land below p often enough: $\Pr[p < \min h(G_\alpha) \mid h(\prec\alpha)] \le (1-p)^{(1-o(1))|G_\alpha|}$. However, a worst-case arrangement of G_α could make all keys equal, which would give the terrible bound of just $1 - p$.

To refine the analysis, we can use Lemma 4, which says that for $d = O(1)$, all groups G_α are d-bounded with probability $\ge 1 - \frac{1}{n^2}$. If G_α is d-bounded, its keys cannot cluster in less than $\lceil |G_\alpha|/d \rceil$ different bins.

When a group G_α has more than one key in some bin, we pick one of them as a *representative*, by some arbitrary (but fixed) tie-breaking rule. Let R_α be the set of representatives of G_α. Observe that the set $R_\alpha \subseteq G_\alpha$ is decided once we condition on $h(\prec\alpha)$. Indeed, the hash codes for keys in G_α are decided up to a shift by $h(\alpha)$, and this common shift cannot change how keys cluster into bins. We obtain:

$$\Pr[p < \min h(G_\alpha) \mid h(\prec\alpha)] \;\le\; \Pr[p < \min h(R_\alpha) \mid h(\prec\alpha)] \;=\; 1 - p|R_\alpha| \;\le\; (1-p)^{|R_\alpha|}$$

To conclude $\Pr[p < \min h(R_\alpha)] = 1 - p|R_\alpha|$ we used that the representatives are in different bins, so at most one can land below p. Remember that $|R_\alpha|$ is a function of $h(\prec\alpha)$. By d-boundedness, $|R_\alpha| \ge |G_\alpha|/d$, so we get $\Pr[p < \min h(G_\alpha) \mid h(\prec\alpha)] \le (1-p)^{|G_\alpha|/d}$ for almost all $h(\prec\alpha)$. Unfortunately, this is a far cry from the desired exponent, $|G_\alpha| \cdot (1 - \widetilde{O}(n^{-1/c}))$.

To get a sharper bound, we will need a dynamic view of the representatives. After fixing $h(\prec\alpha)$, we know whether two keys x and y collide whenever the symmetric difference $x \Delta y = (x \setminus y) \cup (y \setminus x)$ consists only of position-characters $\prec \alpha$. Define $R_\beta(\alpha)$ to be our understanding of the representatives R_β just before character α is revealed: from any subset of G_β that is known to collide, we select only one key.

After the query characters get revealed, we don't know of any collisions yet (we know only one character per position), so $R_\beta(\alpha_0) = G_\beta$. The set of representatives decreases in time, as we learn about more collisions, and $R_\beta(\beta) = R_\beta$ is the final value (revealing β doesn't change the clustering of G_β).

Let $C(\alpha)$ be the number of key pairs (x, y) from the same group G_β ($\beta \succ \alpha$) such that $\alpha = \max_\prec(x \Delta y)$. These are the pairs whose collisions is decided when $h(\alpha)$ is revealed, since $h(\alpha)$ is the last unknown hash code in the keys, besides the common ones. Let α^+ be the successor of α in the order $\prec$. Consider the total number of representatives before and after $h(\alpha)$ is revealed: $\sum_\beta |R_\beta(\alpha)|$ versus $\sum_\beta |R_\beta(\alpha^+)|$. The maximum change between these quantities is $\leq C(\alpha)$, while the expected change is $\leq C(\alpha) \cdot \frac{\ell}{n}$. This is because $h(\alpha)$ makes every pair (x, y) collide with probability $\frac{\ell}{n}$, regardless of the previous hash codes in $(x \Delta y) \setminus \{\alpha\}$. Note, however, that the number of colliding pairs may overestimate the decrease in the representatives if the same key is in multiple pairs.

Let $n(\succ \alpha) = \sum_{\beta \succ \alpha} |G_\alpha|$ and define $n(\succeq \alpha)$ simmilarly. Our main inductive claim is:

Lemma 9. *For any setting $h(\prec \alpha)$ such that $h(q) = p$ and $\sum_{\beta \succeq \alpha} |R_\beta(\alpha)| = r$, we have:*

$$\Pr\left[\left(p < \min \bigcup_{\beta \succeq \alpha} h(G_\beta)\right) \wedge (\forall \alpha) G_\alpha \text{ } d\text{-bounded} \mid h(\prec \alpha)\right] \leq P(\alpha, p, r)$$

where we define $P(\alpha, p, r) = (1-p)^r + (1-p)^{n(\succeq \alpha)/(2d)} \cdot \sum_{\beta \succeq \alpha} \frac{4C(\beta) \cdot (\ell/n)}{n(\succ \beta)/d}$.

As the definition $P(\alpha, p, r)$ may look intimidating, we first try to demystify it, while giving a sketch for the lemma's proof (the formal proof appears in §3.3.) The lemma looks at the worst-case probability, over prior choices $h(\prec \alpha)$, that $p = h(q)$ remains the minimum among groups $G_\alpha, G_{\alpha^+}, \ldots$. After seeing the prior hash codes, the number of representatives in these groups is $r = \sum_{\beta \succeq \alpha} |R_\beta(\alpha)|$. In the ideal case when $h(\alpha), h(\alpha^+), \ldots$ do not introduce any additional collisions, we have r representatives that could beat p for the minimum. As argued above, the probability that p is smaller than all these representatives is $\leq (1-p)^r$. Thus, the first term of $P(\alpha, p, r)$ accounts for the ideal case when no more collisions occur.

On the other hand, the factor $(1-p)^{n(\succeq \alpha)/(2d)}$ accounts for the worst case, with no guarantee on the representatives except that the groups are d-bounded (the 2 in the exponent is an artifact). Thus, $P(\alpha, p, r)$ interpolates between the best case and the worst case. This is explained by a convexity argument: the bound is maximized when $h(\alpha)$ mixes among two extreme strategies — it creates no more collisions, or creates the maximum it could.

It remains to understand the weight attached to the worst-case probability. After fixing $h(\alpha)$, the maximum number of remaining representatives is $\hat{r} = \sum_{\beta \succ \alpha} |R_\beta(\alpha)|$. The expected number is $\geq \hat{r} - C(\alpha)\frac{\ell}{n}$, since every collision happens with probability $\frac{\ell}{n}$. By a Markov bound, the worst case (killing most representatives) can only happen with probability $O(\frac{\ell}{n} C(\alpha)/\hat{r})$. The weight of the worst case follows by $\hat{r} \geq n(\succ \alpha)/d$ and letting these terms accrue in the induction for $\beta \succ \alpha$.

Deriving the upper bound. We now prove the upper bound on $\Pr[h(q) < h(S)]$ assuming Lemma 9. Let α_0 be the first position-character fixed after the query. Since fixing the query cannot eliminate representatives,

$$\Pr[p < \min h(S) \wedge (\forall \alpha) G_\alpha \text{ } d\text{-bounded} \mid h(q) = p] \leq P(\alpha_0, p, n)$$

Lemma 10. $P(\alpha_0, p, n) \leq (1-p)^n + (1-p)^{n/(2d)} \cdot \frac{O(\lg^2 n)}{n^{1/c}}$.

Proof. We will prove that $A = \sum_{\beta \succ \alpha_0} \frac{C(\beta)}{n(\succ \beta)} \leq n^{1-1/c} \cdot H_n$, where H_n is the Harmonic number.

Consider all pairs (x, y) from the same group G_γ, and order them by $\beta = \max_\prec(x \Delta y)$. This is the time when the pair gets counted in some $C(\beta)$ as a potential collision. The contribution of the pair to the sum is $1/n(\succ \beta)$, so this contribution is maximized if β immediately precedes γ in the order $\prec$. That is, the sum is maximized when $C(\beta) = \binom{|G_{\beta^+}|}{2}$. We obtain $A \leq \sum_\beta \frac{|G_\beta|^2}{2} / n(\succeq \beta) \leq n^{1-1/c} \cdot \sum_\beta |G_\beta| / n(\succeq \beta)$. In this sum, each key $x \in G_\beta$ contributes $1/n(\succeq \beta)$, which is bounded by one over the number of keys following x. Thus $A \leq H_n$. □

To achieve our original goal, bounding $\Pr[h(q) < h(S)]$, we proceed as follows:

$$\begin{aligned} \Pr[h(q) < h(S)] &\leq \tfrac{1}{n^2} + \int_0^{\ell/n} \Pr[p < \min h(S) \mid h(q) = p] \mathrm{d}p \\ &\leq \tfrac{1}{n^2} + \Pr[(\exists \alpha) G_\alpha \text{ not } d\text{-bounded}] + \int_0^{\ell/n} P(\alpha_0, p, n) \mathrm{d}p \end{aligned}$$

By Lemma 4, all groups are d-bounded with probability $1 - \frac{1}{n^2}$. We also have

$$\int_0^{\ell/n} (1-p)^n \mathrm{d}p = \left. \frac{-(1-p)^{n+1}}{n+1} \right|_{p=0}^{\ell/n} \leq \frac{1}{n+1}$$

Thus:

$$\Pr[h(q) < h(S)] \leq \frac{O(1)}{n^2} + \frac{1}{n+1} + \frac{1}{n/(2d)+1} \cdot \frac{O(\lg^2 n)}{n^{1/c}} = \frac{1}{n} \cdot \left(1 + \frac{O(\lg^2 n)}{n^{1/c}}\right).$$

3.3 Proof of Lemma 9

Recall that we are fixing some choice of $h(\prec \alpha)$ and bounding:

$$A = \Pr\left[p < \min \bigcup_{\beta \succeq \alpha} h(G_\beta) \ \wedge \ (\forall \alpha) G_\alpha \ d\text{-bounded} \mid h(\prec \alpha)\right]$$

If for some β, $|R_\beta(\alpha)| < |G_\beta|/d$, it means not all groups are d-bounded, so $A = 0$. If all groups are d-bounded and we finished fixing all position-characters, $A = 1$. These form the base cases of our induction.

The remainder of the proof is the inductive step. We first break the probability into:

$$A_1 \cdot A_2 = \Pr\left[p < \min h(G_\alpha) \mid h(\prec \alpha)\right] \cdot \Pr\left[\bigcup_{\beta \succ \alpha} h(G_\beta) \wedge (\forall \alpha) G_\alpha \ d\text{-bounded} \mid h(\prec \alpha), p > \min h(G_\alpha)\right]$$

As $h(\alpha)$ is uniformly random, each representative in R_α has a probability of p of landing below p. These events are disjoint because p is in the minimum bin, so $A_1 = 1 - p \cdot |R_\alpha| \leq (1-p)^{|R_\alpha|}$.

After using R_α, we are left with $\hat{r} = r - |R_\alpha| = \sum_{\beta \succ \alpha} |R_\beta(\alpha)|$ representatives. After $h(\alpha)$ is chosen, some of the representative of $\hat{r}$ are lost. Define the random variable $\Delta = \sum_{\beta \succ \alpha} \left(|R_\beta(\alpha)| - |R_\beta(\alpha^+)|\right)$ to measure this loss.

Let $\Delta^{\max} \geq \hat{r} - \frac{n(\succ \alpha)}{d}$ be a value to be determined. We only need to consider $\Delta \leq \Delta^{\max}$. Indeed, if more than $\Delta^{\max}$ representatives are lost, we are left with less than $n(\succ \alpha)/d$ representatives, so some group is not d-bounded, and the probability is zero. We can now bound A_2 by the induction hypothesis:

$$A_2 \leq \sum_{\delta=0}^{\Delta^{\max}} \Pr[\Delta = \delta \mid h(\prec \alpha), p > \min h(G_\alpha)] \cdot P(\alpha^+, p, \hat{r} - \delta)$$

where we had $P(\alpha^+, p, \hat{r} - \delta) = (1-p)^{\hat{r}-\delta} + (1-p)^{n(\succ\alpha)/(2d)} \cdot \sum_{\beta \succ \alpha} \frac{4C(\beta) \cdot (\ell/n)}{n(\succ\beta)/d}$.

Observe that the second term of $P(\alpha^+, p, \hat{r} - \delta)$ does not depend on δ so:

$$A_2 \leq A_3 + (1-p)^{n(\succ\alpha)/(2d)} \cdot \sum_{\beta \succ \alpha} \frac{4C(\beta) \cdot (\ell/n)}{n(\succ\beta)/d}$$

where $A_3 = \sum_{\delta=0}^{\Delta^{\max}} \Pr[\Delta = \delta \mid h(\prec\alpha), p > \min h(G_\alpha)] \cdot (1-p)^{\hat{r}-\delta}$.

It remains to bound A_3. We observe that $(1-p)^{\hat{r}-\delta}$ is convex in δ, so its achieves the maximum value if all the probability mass of Δ is on 0 and $\Delta^{\max}$, subject to preserving the mean.

Observation 11. *We have:* $\mathbf{E}[\Delta \mid h(\prec\alpha), p > \min h(G_\alpha)] \leq 2 \cdot C(\alpha) \cdot \frac{\ell}{n}$.

Proof. As discussed earlier, a representative disappears when we have a pair $x, y \in R_\beta(\alpha)$ that lands in the same bin due to $h(\alpha)$. This can only happen if (x, y) is counted in $C(\alpha)$, i.e. $\alpha = \max_\prec(x \Delta y)$. If $h(\alpha)$ is uniform, such a pair (x, y) collides with probability $\frac{\ell}{n}$, regardless of $h(\prec\alpha)$. By linearity of expectation $\mathbf{E}[\Delta \mid h(\prec\alpha)] \leq C(\alpha) \cdot \frac{\ell}{n}$.

However, we have to condition on the event $p > \min h(G_\alpha)$, which makes $h(\alpha)$ non-uniform. Since $p < \frac{\ell}{n}$ and $|G_\alpha| \leq n^{1-1/c}$, we have $\Pr[p < \min h(G_\alpha)] < 1/2$. Therefore, conditioning on this event can at most double the expectation of positive random variables. □

A bound on A_3 can be obtained by assuming $\Pr[\Delta = \Delta^{\max}] = \left(2 \cdot C(\alpha) \cdot \frac{\ell}{n}\right) / \Delta^{\max}$, and all the rest of the mass is on $\Delta = 0$. This gives:

$$A_3 \leq (1-p)^{\hat{r}} + \frac{2 \cdot C(\alpha) \cdot (\ell/n)}{\Delta^{\max}} \cdot (1-p)^{\hat{r}-\Delta^{\max}}$$

Remember that we promised to choose $\Delta^{\max} \geq \hat{r} - \frac{n(\succ\alpha)}{d}$. We now fix $\Delta^{\max} = \hat{r} - \frac{n(\succ\alpha)}{2d}$. We are guaranteed that $\hat{r} \geq \frac{n(\succ\alpha)}{d}$, since otherwise some group is not d-bounded. This means $\Delta^{\max} \geq \frac{n(\succ\alpha)}{2d}$. We have obtained a bound on A_3:

$$\begin{aligned}
A_3 &\leq (1-p)^{\hat{r}} + \frac{2 \cdot C(\alpha) \cdot (\ell/n)}{n(\succ\alpha)/(2d)} \cdot (1-p)^{n(\succ\alpha)/(2d)} \\
\implies A_2 &\leq (1-p)^{\hat{r}} + (1-p)^{n(\succ\alpha)/(2d)} \cdot \sum_{\beta \succeq \alpha} \frac{4C(\beta) \cdot (\ell/n)}{n(\succ\beta)/d} \\
\implies A &\leq (1-p)^{|R_\alpha|} \cdot (1-p)^{r-|R_\alpha|} + (1-p)^{n(\succ\alpha)/(2d)} \cdot \sum_{\beta \succeq \alpha} \frac{4C(\beta) \cdot (\ell/n)}{n(\succ\beta)/d}
\end{aligned}$$

This completes the proof of Lemma 9, and the bound on minwise independence.

4 Analysis of Cuckoo Hashing

We begin with the negative side of our result:

Observation 12. *There exists a set S of n keys such that cuckoo hasing with simple tabulation hashing cannot place S into two tables of size $2n$ with probability $\Omega(n^{-1/3})$.*

Proof. The hard instance is the 3-dimensional cube $[n^{1/3}]^3$. Here is a sufficient condition for cuckoo hashing to fail:

- there exist $a, b, c \in [n^{1/3}]^2$ with $h_0(a) = h_0(b) = h_0(c)$;
- there exist $x, y \in [n^{1/3}]$ with $h_1(x) = h_1(y)$.

If both happen, then the elements ax, ay, bx, by, cx, cy cannot be hashed. Indeed, on the left side $h_0(a) = h_0(b) = h_0(c)$ so they only occupy 2 positions. On the right side, $h_1(x) = h_1(y)$ so they only occupy 3 positions. In total they occupy $5 < 6$ positions.

The probability of 1. is asymptotically $(n^{2/3})^3/n^2 = \Omega(1)$. This is because tabulation (on two characters) is 3-independent. The probability of 2. is asymptotically $(n^{1/3})^2/n = \Omega(1/n^{1/3})$. So overall cuckoo hashing fails with probability $\Omega(n^{-1/3})$. □

Our positive result will effectively show that this is the worst possible instance: for any set S, the failure probability is $O(n^{-1/3})$.

The proof is an encoding argument. A tabulation hash function from $\Sigma^c \mapsto [m]$ has entropy $|\Sigma|^c \lg m$ bits; we have two random functions h_0 and h_1. If, under some event $\mathcal{E}$, one can encode the two hash functions h_0, h_1 using $\left(2|\Sigma|^c \lg m\right) - \gamma$ bits, it follows that $\Pr[\mathcal{E}] = O(2^{-\gamma})$. Letting $\mathcal{E}_S$ denote the event that cuckoo hashing fails on the set of keys S, we will demonstrate a saving of $\gamma = \frac{1}{3}\lg n - f(c, \varepsilon) = \frac{1}{3}\lg n - O(1)$ bits in the encoding. Note that we are analyzing simple tabulation on a *fixed* set of n keys, so both the encoder and the decoder know S.

We will consider various cases, and give algorithms for encoding some subset of the hash codes (we can afford $O(1)$ bits in the beginning of the encoding to say which case we are in). At the end, the encoder will always list all the remaining hash codes in order. If the algorithm chooses to encode k hash codes, it will use space at most $k \lg m - \frac{1}{3}\lg n + O(1)$ bits. That is, it will save $\frac{1}{3}\lg n - O(1)$ bits in the complete encoding of h_0 and h_1.

4.1 An easy way out

A *subkey* is a set of position-characters on distinct positions. If a is a subkey, we let $C(a) = \{x \in S \mid a \subseteq x\}$ be the set of "completions" of a to a valid key.

We first consider an easy way out: there subkeys a and b on the positions such that $|C(a)| \geq n^{2/3}$, $|C(b)| \geq n^{2/3}$, and $h_i(a) = h_i(b)$ for some $i \in \{0, 1\}$. Then we can easily save $\frac{1}{3}\lg n - O(1)$ bits. First we write the set of positions of a and b, and the side of the collision ($c+1$ bits). There are at most $n^{1/3}$ subkeys on those positions that have $\geq n^{2/3}$ completions each, so we can write the identities of a and b using $\frac{1}{3}\lg n$ bits each. We write the hash codes h_i for all characters in $a \Delta b$ (the symmetric difference of a and b), skipping the last one, since it can be deduced from the collision. This uses $c+1+2\cdot\frac{1}{3}\lg n + (|a\Delta b|-1)\lg m$ bits to encode $|a\Delta b|$ hash codes, so it saves $\frac{1}{3}\lg n - O(1)$ bits.

The rest of the proof assumes that there is no easy way out.

4.2 Walking Along an Obstruction

Consider the bipartite graph with m nodes on each side and n edges going from $h_0(x)$ to $h_1(x)$ for all $x \in S$. Remember that cuckoo hashing succeeds if and only if no component in this graph has more edges than nodes. Assuming cuckoo hashing failed, the encoder can find a subgraph with one of two possible obstructions: (1) a cycle with a chord; or (2) two cycles connected by a path (possibly a trivial path, i.e. the cycles simply share a vertex).

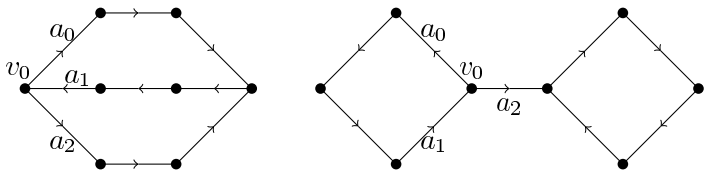


Figure 1: Minimal obstructions to cuckoo hashing.

Let v_0 be a node of degree 3 in such an obstruction, and let its incident edges be a_0, a_1, a_2. The obstruction can be traversed by a walk that leaves v_0 on edge a_0, returns to v_0 on edge a_1, leaves again on a_2, and eventually meets itself. Other than visting v_0 and the last node twice, no node or edge is repeated. See Figure 1.

Let $x_1, x_2, \ldots$ be the sequence of keys in the walk. The first key is $x_1 = a_0$. Technically, when the walk meets itself at the end, it is convenient to expand it with an extra key, namely the one it first used to get to the meeting point. This repeated key marks the end of the original walk, and we chose it so that it is not identical to the last original key. Let $x_{\le i} = \bigcup_{j \le i} x_j$ be the position-characters seen in keys up to x_i. Define $\hat{x}_i = x_i \setminus x_{<i}$ to be the position-characters of x_i not seen previously in the sequence. Let k be the first position such that $\hat{x}_{k+1} = \emptyset$. Such a k certainly exists, since the last key in our walk is a repeated key.

At a high level, the encoding algorithm will encode the hash codes of $\hat{x}_1, \ldots, \hat{x}_k$ in this order. Note that the obstruction, hence the sequence (x_i), depends on the hash functions h_0 and h_1. Thus, the decoder does not know the sequence, and it must also be written in the encoding.

For notational convenience, let $h_i = h_{i \bmod 2}$. This means that in our sequence x_i and x_{i+1} collide in their h_i hash code, that is $h_i(x_i) = h_i(x_{i+1})$. Formally, we define 3 subroutines:

ID(x)**:** Write the identity of $x \in S$ in the encoding, which takes $\lg n$ bits.

HASHES(h_i, x_k)**:** Write the hash codes h_i of the characters $\hat{x}_k$. This takes $|\hat{x}_k| \lg m$ bits.

COLL(x_i, x_{i+1})**:** Document the collision $h_i(x_i) = h_i(x_{i+1})$. We write all h_i hash codes of characters $\hat{x}_i \cup \hat{x}_{i+1}$ in some fixed order. The last hash code of $\hat{x}_i \Delta \hat{x}_{i+1}$ is redundant and will be ommitted. Indeed, the decoder can compute this last hash code from the equality $h_i(x_i) = h_i(x_{i+1})$. Since $\hat{x}_{i+1} = x_{i+1} \setminus x_{\le i}$, $\hat{x}_{i+1} \setminus \hat{x}_i \neq \emptyset$, so there exists a hash code in $\hat{x}_i \Delta \hat{x}_{i+1}$. This subroutine uses $\left(|\hat{x}_i \cup \hat{x}_{i+1}| - 1\right) \lg m$ bits, saving $\lg m$ bits compared to the trivial alternative: HASHES(h_i, x_i); HASHES(h_i, x_{i+1}).

To decode the above information, the decoder will need enough context to synchronize with the coding stream. For instance, to decode COLL(x_i, x_{i+1}), one typically needs to know i, and the identities of x_i and x_{i+1}.

Our encoding begins with the value k, encoded with $O(\lg k)$ bits, which allows the decoder to know when to stop. The encoding proceeeds with the output of the stream of operations:

$$\text{ID}(x_1); \text{HASHES}(h_0, x_1); \text{ID}(x_2); \text{COLL}(x_1, x_2); \ldots \text{ID}(x_k); \text{COLL}(x_k, x_{k-1}); \text{HASHES}(h_k, x_k)$$

We observe that for each $i > 1$, we save ε bits of entropy. Indeed, ID(x_i) uses $\lg n$ bits, but COLL(x_{i-1}, x_i) then saves $\lg m = \lg((1+\varepsilon)n) \ge \varepsilon + \lg n$ bits.

The trouble is ID(x_1), which has an upfront cost of $\lg n$ bits. We must devise algorithms that modify this stream of operations and save $\frac{4}{3} \lg n - O(1)$ bits, giving an overall saving of $\frac{1}{3} \lg n - O(1)$. (For intuition, observe that a saving that ignores the cost of ID(x_1) bounds the probability of an obstruction at some fixed vertex in the graph. This probability must be much smaller than $1/n$, so we can union bound over all vertices. In encoding terminology, this saving must be much more than $\lg n$ bits.)

We will use modifications to all types of operations. For instance, we will sometimes encode $\textsc{Id}(x)$ with much less than $\lg n$ bits. At other times, we will be able to encode $\textsc{Coll}(x_i, x_{i+1})$ with the cost of $|\hat{x}_i \cup \hat{x}_{i+1}| - 2$ characters, saving $\lg n$ bits over the standard encoding.

Since we will make several such modifications, it is crucial to verify that they only touch distinct operations in the stream. Each modification to the stream will be announced at the beginning of the stream with a pointer taking $O(\lg k)$ bits. This way, the decoder knows when to apply the special algorithms. We note that terms of $O(\lg k)$ are negligible, since we are already saving εk bits by the basic encoding (ε bits per edge). For any k, $O(\lg k) \leq \varepsilon k + f(c, \varepsilon) = k + O(1)$. Thus, if our overall saving is $\frac{1}{3}\lg n - O(\lg k) + \varepsilon k$, it achieves the stated bound of $\lg n - O(1)$.

4.3 Safe Savings

Remember that $\hat{x}_{k+1} = \emptyset$, which suggests that we can save a lot by local changes towards the end of the encoding. We have $x_{k+1} \subset x_{\leq k}$, so $x_{k+1} \setminus x_{<k} \subseteq \hat{x}_k$. We will first treat the case when $x_{k+1} \setminus x_{<k}$ is a proper subset of $\hat{x}_k$ (including the empty subset). This is equivalent to $\hat{x}_k \not\subset x_{k+1}$.

Lemma 13 (safe-strong)**.** *If $\hat{x}_k \not\subset x_{k+1}$, we can save $\lg n - O(c \lg k)$ bits by changing $\textsc{Hashes}(x_k)$.*

Proof. We can encode $\textsc{Id}(x_{k+1})$ using $c \lg k$ extra bits, since it consists only of known characters from $x_{\leq k}$. For each position $1 \ldots c$, it suffices to give the index of a previous x_i that contained the same position-character. Then, we will write all hash codes h_k for the characters in $\hat{x}_k$, except for some $\alpha \in \hat{x}_k \setminus x_{k+1}$. From $h_k(x_k) = h_k(x_{k+1})$, we have $h_k(\alpha) = h_k(x_k \setminus \{\alpha\}) \oplus h_k(x_{k+1})$. All quantities on the right hand side are known (in particular $\alpha \notin x_{k+1}$), so the decoder can compute $h_k(\alpha)$. □

It remains to treat the case when the last revealed characters of x_{k+1} are precisely $\hat{x}_k$: $\hat{x}_k \subset x_{k+1}$. That is, both x_k and x_{k+1} consist of $\hat{x}_k$ and some previously known characters. In this case, the collision $h_k(x_k) = h_k(x_{k+1})$ does not provide us any information, since it reduces to the trivial $h_k(\hat{x}_k) = h_k(\hat{x}_k)$. Assuming that we didn't take the "easy way out", we can still guarantee a more modest saving of $\frac{1}{3}\lg n$ bits:

Lemma 14 (safe-weak)**.** *Let K be the set of position-characters known before encoding $\textsc{Id}(x_i)$, and assume there is no easy way out. If $x_i \Delta x_{i+1} \subseteq x_{<i}$, then we can encode both $\textsc{Id}(x_i)$ and $\textsc{Id}(x_{i+1})$ using a total of $\frac{2}{3}\lg n + O(c \lg |K|)$ bits.*

A typical case where we apply the lemma is $i = k$ and $K = x_{<k}$. If $\hat{x}_k \subset x_{k+1}$, we have $x_k \Delta x_{k+1} \subset K$. Thus, we can obtain $\textsc{Id}(x_k)$ for roughly $\frac{2}{3}\lg n$ bits, which saves $\frac{1}{3}\lg n$ bits.

of Lemma 14. With $O(c \lg k)$ bits, we can code the subkeys $x_i \cap x_{<i}$ and $x_{i+1} \cap x_{<i}$. It remains to code $z = x_i \setminus x_{<i} = x_{i+1} \setminus x_{<i}$. Since z is common to both keys x_i and x_{i+1}, we have that $x_i \setminus z$ and $x_{i+1} \setminus z$ are subkeys on the same positions. With no easy way out and $h_i(x_i \setminus z) = h_i(x_{i+1} \setminus z)$, we must have $|C(x_i \setminus z)| \leq n^{2/3}$ or $|C(x_{i+1} \setminus z)| \leq n^{2/3}$. In the former case, we code z as a member of $C(x_i \setminus z)$ with $\lceil \frac{2}{3}\lg n \rceil$ bits; otherwise we code z as member of $C(x_{i+1} \setminus z)$. □

4.4 Piggybacking

Before moving forward, we present a general situation when we can save $\lg n$ bits by modifying a $\textsc{Coll}(x_i, x_{i+1})$ operation:

Lemma 15. *We can save* $\lg n - O(\lg k)$ *bits by modifying* $\textsc{Coll}(x_i, x_{i+1})$ *if we have identified two (sub)keys* e *and* f *satisfying:*

$$h_i(e) = h_i(f); \qquad e\Delta f \subset x_{\leq i+1}; \qquad \emptyset \neq (e\Delta f) \setminus x_{<i} \neq (x_i \Delta x_{i+1}) \setminus x_{<i}.$$

Proof. In the typical encoding of $\textsc{Coll}(x_i, x_{i+1})$, we saved one redundant character from $h_i(x_i) = h_i(x_{i+1})$, which is an equation involving $(x_i \Delta x_{i+1}) \setminus x_{<i}$ and some known characters from $x_{<i}$. The lemma guarantees a second linearly independent equation over the characters $\hat{x}_i \cup \hat{x}_{i+1}$, so we can save a second redundant character.

Formally, let α be a position-character of $(e\Delta f) \setminus x_{<i}$, and β a position-character in $(x_i \Delta x_{i+1}) \setminus x_{<i}$ but outside $(e\Delta f) \setminus x_{<i}$. Note $\beta \neq \alpha$ and such a β exists by assumption. We write the h_i hash codes of position characters $(\hat{x}_i \cup \hat{x}_{i+1}) \setminus \{\alpha, \beta\}$. The hash $h_i(\alpha)$ can be deduced since α is the last unknown in the equality $h_i(e \setminus f) = h_i(f \setminus e)$. The hash $h_i(\beta)$ can be deduced since it is the last unknown in the equality $h_i(x) = h_i(x_{i+1})$. □

While the safe saving ideas only require simple local modifications to the encoding, they achieve a weak saving of $\frac{1}{3} \lg n$ bits for the case $\hat{x}_k \subset x_{k+1}$. A crucial step in our proof is to obtain a saving of $\lg n$ bits for this case. We do this by one of the following two lemmas:

Lemma 16 (odd-size saving)**.** *Consider two edges* e, f *and an* $i \leq k-2$ *satisfying:*

$$h_{i+1}(e) = h_{i+1}(f); \qquad e \setminus x_{\leq i} \neq f \setminus x_{\leq i}; \qquad e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1}.$$

We can save $\lg n - O(c \lg k)$ *bits by changing* $\textsc{Coll}(x_{i+1}, x_{i+2})$.

Proof. We apply Lemma 15 with the subkeys $\tilde{e} = e \setminus f$ and $\tilde{f} = f \setminus e$. We can identify these in $O(c \lg k)$ bits, since they only contain characters of $x_{\leq i+1}$. Since e and f have different free characters before $\hat{x}_{i+1}$, but identical free characters afterwards, it must be that $\tilde{e} \cup \tilde{f} \subset x_{i+1}$ by $\tilde{e} \cup \tilde{f} \not\subseteq x_{\leq i}$. To show $(e\Delta f) \setminus x_{<i} \neq (x_{i+1} \Delta x_{i+2}) \setminus x_{\leq i}$, remark that $\hat{x}_{i+2} \neq \emptyset$ and $\hat{x}_{i+2}$ cannot have characters of $\tilde{e} \cup \tilde{f}$. Thus, Lemma 15 applies. □

Lemma 17 (piggybacking)**.** *Consider two edges* e, f *and an* $i \leq k-1$ *satisfying:*

$$h_i(e) = h_i(f); \qquad e \setminus x_{\leq i} \neq f \setminus x_{\leq i}; \qquad e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1}.$$

We can encode $\textsc{Id}(e)$ *and* $\textsc{Id}(f)$ *using only* $O(c \lg k)$ *bits, after modifications to* $\textsc{Id}(x_i)$, $\textsc{Id}(x_{i+1})$, *and* $\textsc{Coll}(x_i, x_{i+1})$.

The proof of this lemma is more delicate, and is given below. The difference between the two lemmas is the parity (side in the bipartitie graph) of the collision of x_i and x_{i+1} versus the collision of e and f. In the second result, we cannot actually save $\lg n$ bits, but we can encode $\textsc{Id}(e)$ and $\textsc{Id}(f)$ almost for free: we say e and f piggyback on the encodings of x_i and x_{i+1}.

We demonstrate the combination of the two lemmas by saving $\lg n$ bits in the case $\hat{x}_k \subset x_{k+1}$. We will choose $e = x_k$ and $f = x_{k+1}$. We have $e \setminus x_{<k} = f \setminus x_{<k} = \hat{x}_k$. On the other hand, $e \setminus x_1 \neq f \setminus x_1$ since x_1 only reveals one character per position. Thus there must be some $1 \leq i < k-1$ where the transition happens: $e \setminus x_{\leq i} \neq f \setminus x_{\leq i}$ but $e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1}$. If i has the opposite parity compared to k, Lemma 16 saves a $\lg n$ term. (Note that $i \leq k-2$ as required by the lemma.)

If i has the same parity as k, Lemma 17 gives us $\textsc{Id}(x_k)$ at negligible cost. Then, we can remove the operation $\textsc{Id}(x_k)$ from the stream, and save $\lg n$ bits. (Again, note that $i \leq k-2$ as requires.)

Proof of Lemma 17. The lemma assumed $e \setminus x_{\le i} \neq f \setminus x_{\le i}$ but $e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$. Therefore, $e \Delta f \subset x_{\le i+1}$ and $(e \Delta f) \cap \hat{x}_{i+1} \neq \emptyset$. Lemma 15 applies if we furthermore have $(e \Delta f) \setminus x_{<i} \neq (x_i \Delta x_{i+1}) \setminus x_{<i}$. If the lemma applies, we have a saving of $\lg n$, so we can afford to encode $\textsc{Id}(e)$. Then $\textsc{Id}(f)$ can be encoded using $O(c \lg k)$ bits, since f differs from e only in position-characters from $x_{\le i+1}$.

If the lemma does not apply, we have a lot of structure on the keys. Let $y = \hat{x}_i \setminus (e \cup f)$ and $g = e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$. We must have $y \subset x_{i+1}$, for otherwise $\hat{x}_i \setminus x_{i+1}$ contains an elements outside $e \Delta f$ and the lemma applies. We must also have $\hat{x}_{i+1} \subset e \cup f$.

We can write $\textsc{Id}(x_i)$, $\textsc{Id}(x_{i+1})$, $\textsc{Id}(e)$, and $\textsc{Id}(f)$ using $2 \lg n + O(c \lg k)$ bits in total, as follows:

- the coordinates on which y and g appear, taking $2c$ bits.
- the value of y using Huffman coding. Specifically, we consider the projection of all n keys on the coordinates of y. In this distribution, y has frequency $\frac{C(y)}{n}$, so its Huffman code will use $\lg \frac{n}{C(y)} + O(1)$ bits.
- the value of g using Huffman coding. This uses $\lg \frac{n}{C(g)} + O(1)$ bits.
- if $C(y) \le C(g)$, we write x_i and x_{i+1}. Each of these requires $\lceil \log_2 C(y) \rceil$ bits, since $y \subset x_i, x_{i+1}$ and there are $C(y)$ completions of y to a full key. Using an additional $O(c \lg k)$ bits, we can write $e \cap x_{\le i+1}$ and $f \cap x_{\le i+1}$. Remember that we already encoded $g = e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$, so the decoder can recover e and f.
- if $C(g) < C(y)$, we write e and f, each requiring $\lceil \log_2 C(g) \rceil$ bits. Since we know $y = \hat{x}_i \setminus (e \cup f)$, we can write x_i using $O(c \lg k)$ bits: write the old characters outside $\hat{x}_i$, and which positions of $e \cup f$ to reuse in $\hat{x}_i$. We showed $\hat{x}_{i+1} \subset e \cup f$, so we can also write x_{i+1} using $O(c \lg k)$.

Overall, the encoding uses space:

$$\lg \tfrac{n}{C(\xi)} + \lg \tfrac{n}{C(\hat{e}_{i+1})} + 2 \lg \min \left\{ C(\xi), C(\hat{e}_{i+1}) \right\} + O(c \lg k) \;\le\; 2 \lg n + O(c \lg k) \quad \square$$

4.5 Putting it Together

We now show how to obtain a saving of at least $\frac{4}{3} \lg n - O(c \lg k)$ bits by a careful combination of the above techniques. Recall that our starting point is three edges a_0, a_1, a_2 with $h_0(a_0) = h_0(a_1) = h_0(a_2)$. The walk $x_1, ..., x_{k+1}$ started with $x_1 = a_0$ and finished when $\hat{x}_{k+1} = \emptyset$. We will now involve the other starting edges a_1 and a_2. The analysis will split into many cases, each ended by a '$\diamond$'.

Case 1: One of a_1 and a_2 contains a free character. Let $j \in \{1, 2\}$ such that $a_j \not\subseteq x_{\le k}$. Let $y_1 = a_j$. We consider a walk $y_1, y_2, \ldots$ along the edges of the obstruction.Let $\hat{y}_i = y_i \setminus x_{\le k} \setminus y_{\le i}$ be the free characters of y_i (which also takes all x_i's into consideration). We stop the walk the first time we observe $\hat{y}_{\ell+1} = \emptyset$. This must occur, since the graph is finite and there are no leaves (nodes of degree one) in the obstruction. Thus, at the latest the walk stops when it repeats an edge.

We use the standard encoding for the second walk:

$$\textsc{Id}(y_1); \textsc{Coll}(a_0, y_1); \textsc{Id}(y_2); \textsc{Coll}(y_2, y_1); \ldots; \textsc{Id}(y_\ell); \textsc{Coll}(y_{\ell-1}, y_\ell); \textsc{Hashes}(h_\ell, y_\ell)$$

Note that every pair $\textsc{Id}(y_j), \textsc{Coll}(y_{j-1}, y_j)$ saves ε bits, including the initial $\textsc{Id}(y_1), \textsc{Coll}(a_0, y_1)$. To end the walk, we can use one of the safe savings of Lemmas 13 and 14. These give a saving of $\frac{1}{3} \lg n - O(c \lg(\ell + k))$ bits, by modifying only $\textsc{Hashes}(h_\ell, y_\ell)$ or $\textsc{Id}(y_\ell)$. These local changes cannot interfere with the first walk, so we can use any technique (including piggybacking) to save $\lg n - O(c \log k)$ bits from the first walk. We obtain a total saving of $\frac{4}{3} \lg n - O(1)$, as required. $\diamond$

We are left with the situation $a_1 \cup a_2 \subseteq x_{\leq k}$. This includes the case when a_1 and a_2 are actual edges seen in the walk $x_1, \ldots, x_k$.

Let t_j be the first time a_j becomes known in the walk; that is, $a_j \not\subseteq x_{<t_j}$ but $a_j \subseteq x_{\leq t_j}$. By symmetry, we can assume $t_1 \leq t_2$. We begin with two simple cases.

Case 2: For some $j \in \{1,2\}$, t_j is even and $t_j < k$. We will apply Lemma 15 and save $\lg n - O(c \lg k)$ bits by modifying $\textsc{Coll}(x_{t_j}, x_{t_j+1})$. Since $t_j < k$, this does not interact with safe savings at the end of the stream, so we get total saving of at least $\frac{4}{3} \lg n - O(c \lg k)$.

We apply Lemma 15 on the keys $e = a_0$ and $f = a_j$. We must first write $\textsc{Id}(a_j)$, which takes $O(c \lg k)$ bits given $x_{\leq k}$. We have $a_0 \cup a_j \subseteq x_{\leq t_j}$ by definition of t_j. Since $a_j \cap \hat{x}_{t_j} \neq \emptyset$ and $\hat{x}_{t_j+1} \cap (a_j \cup a_0) = \emptyset$, the lemma applies. $\diamond$

Case 3: For some $j \in \{1,2\}$, t_j is odd and $a_j \setminus x_{<t_j-1} \neq \hat{x}_{t_j-1} \Delta \hat{x}_{t_j}$. This assumption is exactly what we need to apply Lemma 15 with $e = a_0$ and $f = a_j$. Note that $h_0(e) = h_0(f)$ and t_j is odd, so the lemma modifies $\textsc{Coll}(x_{t_j-1}, x_{t_j})$. The lemma can be applied in conjunction with any safe saving, since the safe savings only require modifications to $\textsc{Id}(x_k)$ or $\textsc{Hashes}(h_k, x_k)$. $\diamond$

We now deal with two cases when $t_1 = t_2$ (both being odd or even). These require a combination of piggybacking followed by safe-weak savings. Note that in the odd case, we may assume $a_1 \setminus x_{<t-1} = a_2 \setminus x_{<t-1} = \hat{x}_{t-1} \Delta \hat{x}_t$ (due to case 3 above), and in the even case we may assume $t_1 = t_2 = k$ (due to case 2 above).

Case 4: $t_1 = t_2 = t$ is odd and $a_1 \setminus x_{<t-1} = a_2 \setminus x_{<t-1} = \hat{x}_{t-1} \Delta \hat{x}_t$. We first get a_1 and a_2 by piggybacking or odd-side saving. Let i be the largest value such that $a_1 \setminus x_{\leq i} \neq a_2 \setminus x_{\leq i}$. Since $a_1 \setminus x_{<t-1} = a_2 \setminus x_{<t-1}$, we have $i \leq t-3$. The last key that piggybacking or odd-side saving can interfere with is x_{t-2}.

We will now use the safe-weak saving of Lemma 14 to encode $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$. The known characters are $K = x_{<t-1} \cup a_1 \cup a_2$, so $x_{t-1} \Delta x_t \subseteq K$. Lemma 14 codes both $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$ with $\frac{2}{3} \lg n + O(c \lg k)$ bits, which represents a saving of roughly $\frac{4}{3} \lg n$ over the original encoding of the two identities. We don't need any more savings from the rest of the walk after x_t. $\diamond$

Case 5: $t_1 = t_2 = k$ is even. Thus, k is even and the last characters of a_1 and a_2 are only revealed by $\hat{x}_k$.

Lemma 18. *We can save $2 \lg n - O(c \lg k)$ bits by modifying $\textsc{Hashes}(h_k, x_k)$, unless both: (1) $a_1 \cap \hat{x}_k = a_2 \cap \hat{x}_k$; and (2)$\hat{x}_k \setminus x_{k+1}$ is the empty set or equal to $a_1 \cap \hat{x}_k$.*

Proof. The h_0 hash codes of the following 3 subkeys are known from the hash codes in $x_{<k}$: $a_1 \cap \hat{x}_k$, $a_2 \cap \hat{x}_k$ (both because we know $h_0(a_0) = h_0(a_1) = h_0(a_2)$), and $\hat{x}_k \setminus x_{k+1}$ (since x_k and x_{k+1} collide). If two of these subsets are distinct and nonempty, we can choose two characters α and β from their symmetric difference. We can encode all characters of $\hat{x}_k$ except for α and β, whose hash codes can be deduced for free.

Since $a_j \cap \hat{x}_k \neq$ in the current case, the situations when we can find two distinct nonempty sets are: (1) $a_1 \cap \hat{x}_k \neq a_2 \cap \hat{x}_k$; or (2) $a_1 \cap \hat{x}_k = a_2 \cap \hat{x}_k$ but $\hat{x}_k \setminus x_{k+1}$ is nonempty and different from them. $\square$

From now on assume the lemma fails. We can still save $\lg n$ bits by modifying $\textsc{Hashes}(h_k, x_k)$. We reveal all hash codes of $\hat{x}_k$, except for one position-character $\alpha \in a_1 \cap \hat{x}_k$. We then specify $\textsc{Id}(a_1)$, which takes $O(c \lg k)$ bits. The hash $h_0(\alpha)$ can then be deduced from $h_0(a_1) = h_0(a_0)$.

We will now apply piggybacking or odd-side saving to a_1 and a_2. Let i be the largest value with $a_1 \setminus x_{\le i} \neq a_2 \setminus x_{\le i}$. Note that $a_1 \setminus x_{<k} = a_2 \setminus x_{<k}$, so $i < k-1$. If i is odd, Lemma 16 (odd-side saving) can save $\lg n$ bits by modifying $\textsc{Coll}(x_{i+1}, x_{i+2})$; this works since $i+2 \le k$. If i is even, Lemma 17 (piggybacking) can give use $\textsc{Id}(a)$ and $\textsc{Id}(b)$ at a negligible cost of $O(c \lg k)$ bits. This doesn't touch anything later than $\textsc{Id}(x_{i+1})$, where $i+1 < k$.

When we arrive at $\textsc{Id}(x_k)$, we know the position characters $K = x_{<k} \cup a_1 \cup a_2$. This means that $x_k \Delta x_{k+1} \subseteq K$, because $\hat{x}_k \setminus x_{k+1}$ is either empty or a subset of a_1. Therefore, we can use weak-safe savings from Lemma 14 to code $\textsc{Id}(x_k)$ in just $\frac{1}{3}\lg n + O(c \lg k)$ bits. In total, we have save at least $\frac{4}{3}\lg n - O(c \lg k)$ bits. ◇

It remains to deal with distinct t_1, t_2, i.e. $t_1 < t_2 \le k$. If one of the numbers is even, it must be $t_2 = k$, and then t_1 must be odd (due to case 2). By Case 3, if t_j is odd, we also know $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1} \Delta \hat{x}_{t_j}$. Since these cases need to deal with at least one odd t_j, the following lemma will be crucial:

Lemma 19. *If $t_j \le k$ is odd and $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1} \Delta \hat{x}_{t_j}$, we can code $\textsc{Id}(x_{t_j-1})$ and $\textsc{Id}(x_{t_j})$ with $\frac{3}{2}\lg n + O(c \lg k)$ bits in total.*

Proof. Consider the subkey $y = \hat{x}_{t_j-1} \setminus x_{t_j}$. We first specify the positions of y using c bits. If $C(y) \ge \sqrt{n}$, there are at most $\sqrt{n}$ possible choices of y, so we can specify y with $\frac{1}{2}\lg n$ bits. We can also identify x_{t_j} with $\lg n$ bits. Then $\textsc{Id}(x_{t_j-1})$ requires $O(c \lg k)$ bits, since $x_{t_j-1} \subseteq y \cup x_{t_j} \cup x_{<t_j-1}$.

If $C(y) \le \sqrt{n}$, we first specity $\textsc{Id}(x_{t_j-1})$ with $\lg n$ bits. This gives us the subkey $y \subseteq x_{t_j-1}$. Since $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1} \Delta \hat{x}_{t_j}$, it follows that $y \subset a_j$. Thus, we can write $\textsc{Id}(a_j)$ using $\lg C(y) \le \lg \frac{1}{2}\lg n$ bits. Since $x_{t_j} \subseteq x_{\le t_j-1} \cup a_j$, we get $\textsc{Id}(x_{t_j})$ for an additional $O(c \lg k)$ bits. □

Case 6: Both t_1 and t_2 are odd, $t_1 < t_2 < k$, and for all $j \in \{1,2\}$, $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1} \Delta \hat{x}_{t_j}$. We apply Lemma 19 for both $j=1$ and $j=2$, and save $\lg n$ bits in coding $\textsc{Id}(x_{t_1-1})$, $\textsc{Id}(x_{t_1})$, $\textsc{Id}(x_{t_2-1})$, and $\textsc{Id}(x_{t_2})$. These are all distinct keys, because $t_1 < t_2$ and both are odd. Since $t_2 < k$, we can combine this with any safe saving. ◇

Case 7: $t_2 = k$ is even and $t_1 < k$ is odd with $a_1 \setminus x_{<t_1-1} = \hat{x}_{t_1-1} \Delta \hat{x}_{t_1}$. We apply Lemma 19 for $j=1$, and save $\frac{1}{2}\lg n - O(c \lg k)$ bits in coding $\textsc{Id}(x_{t_1-1})$, $\textsc{Id}(x_{t_1})$. We also save $\lg n$ bits by modifying $\textsc{Hashes}(h_0, x_k)$. We reveal all hash codes of $\hat{x}_k$, except for one position-character $\alpha \in a_2 \cap \hat{x}_k$ (which is a nonempty set since $t_2 = k$). We then specify $\textsc{Id}(a_2)$, which takes $O(c \lg k)$ bits. The hash $h_0(\alpha)$ can then be deduced from $h_0(a_2) = h_0(a_0)$. ◇

Case 8: Both t_1 and t_2 are odd, $t_1 < t_2 = k$, and for all $j \in \{1,2\}$, $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1} \Delta \hat{x}_{t_j}$. To simplify notation, let $t_1 = t$. This case is the most difficult. If we can apply strong-safe saving as in Lemma 13, we save $\lg n$ by modifying $\textsc{Hashes}(h_k, x_k)$. We also save $\lg n$ by two applications of Lemma 19, coding $\textsc{Id}(x_{t-1})$, $\textsc{Id}(x_t)$, $\textsc{Id}(x_{k-1})$, and $\textsc{Id}(x_k)$. These don't interact since $t < k$ and both are odd.

The strong-safe saving fails if $\hat{x}_k \subset x_{k+1}$. We will attempt to piggyback for x_k and x_{k+1}. Let i be the largest value such that $x_k \setminus x_{\le i} \neq x_{k+1} \setminus x_{\le i}$. If i is even, we get an odd-side saving of $\lg n$ (Lemma 16). Since this does not affect any identities, we can still apply Lemma 19 to save $\frac{1}{2}\lg n$ on the identities $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$.

Now assume i is odd. We have real piggybacking, which may affect the coding of $\textsc{Id}(x_i)$, $\textsc{Id}(x_{i+1})$ and $\textsc{Id}(x_k)$. Since both i and t are odd, there is at most one common key between $\{x_i, x_{i+1}\}$ and $\{x_{t-1}, x_t\}$. We consider two cases:

- Suppose $x_{t-1} \notin \{x_i, x_{i+1}\}$. Let $y = \hat{x}_{t-1} \setminus x_t$. After piggybacking, which in particular encodes x_t, we can encode $\text{ID}(x_{t-1})$ in $\lg \frac{n}{C(y)} + O(c \lg k)$ bits. Indeed, we can write the positions of y with c bits and then the identity of y using Huffman coding for all subkeys on those positions. Finally the identity of x_{t-1} can be written in $O(c \lg k)$ bits, since $x_{t-1} \subset x_{<t-1} \cup y \cup x_t$.
- Suppose $x_t \notin \{x_i, x_{i+1}\}$. Let $y = \hat{x}_t \setminus x_{t-1}$. As above, we can write $\text{ID}(x_t)$ using $\lg \frac{n}{C(y)} + O(c \lg k)$ bits, after piggybacking.

If $C(y) \geq n^{1/3}$, we have obtained a total saving of $\frac{4}{3} \lg n - O(c \lg k)$: a logarithmic term for $\text{ID}(x_k)$ from piggybacking, and $\frac{1}{3} \lg n$ for $\text{ID}(x_{t-1})$ or $\text{ID}(x_t)$.

Now assume that $C(y) \leq n^{1/3}$. In this case, we do *not* use piggybacking. Instead, we use a variation of Lemma 19 to encode $\text{ID}(x_{t-1})$ and $\text{ID}(x_t)$. First we code the one containing y with $\lg n$ bits. Since $a_1 \setminus x_{<t-1} = \hat{x}_{t-1} \Delta \hat{x}_t$, and therefore $y \subset a_1$, we have $y \subset a_1$. We code $\text{ID}(a_1)$ with $\lg C(y) \leq \frac{1}{3} \lg n$ bits. We obtain the other key among x_{t-1} and x_t using $O(c \lg k)$ bits, since all its characters are known. Thus we have coded $\text{ID}(x_{t-1})$ and $\text{ID}(x_t)$ with $\frac{4}{3} \lg n + O(c \lg k)$ bits, for a saving of roughly $\frac{2}{3} \lg n$ bits.

Next we consider the coding of $\text{ID}(x_{k-1})$ and $\text{ID}(x_k)$. We know that $a_2 \setminus x_{<k-1} = \hat{x}_{k-1} \Delta \hat{x}_k$ and $\hat{x}_k \subset x_{k+1}$. Lemma 19 would guarantee a saving of $\frac{1}{2} \lg n$ bits. However, we will perform an analysis like abobe, obtaining a saving of $\frac{2}{3} \lg n$ bits.

Let $y = \hat{x}_{k-1} \setminus x_k$. First assume $C(y) \geq n^{1/3}$. We use the safe-weak saving of Lemma 14 to encode $\text{ID}(x_k)$ using $\frac{2}{3} \lg n$ bits. We then encode the subkey y using $\lg \frac{n}{C(y)} + O(c) \leq \frac{2}{3} \lg n + O(c)$ bits, and finally x_{k-1} using $O(c \lg k)$ bits. This obtains both $\text{ID}(x_{k-1})$ and $\text{ID}(x_k)$ using $\frac{4}{3} \lg n + O(c \lg k)$ bits.

Now assume $C(y) \leq n^{1/3}$. We first code $\text{ID}(x_{k-1})$ using $\lg n$ bits. This gives us y for the price of c bits. But $a_2 \setminus x_{<k-1} = \hat{x}_{k-1} \Delta \hat{x}_k$, so $y \subset a_2$, and we can code $\text{ID}(a_2)$ using $\lg C(y) \leq \frac{1}{3} \lg n$ bits. Then $\text{ID}(x_k)$ can be coded with $O(c \lg k)$ bits. Again, we obtain both $\text{ID}(x_{k-1})$ and $\text{ID}(x_k)$ for the price of $\frac{4}{3} \lg n + O(c \lg k)$ bits. ◇

This completes our analysis of cuckoo hashing.

5 Linear probing

We consider linear probing using simple tabulation hashing and an array of size m (as in the rest of our analyses, m is a power of two). Let $\alpha = 1 - \varepsilon = \frac{n}{m}$ be the fill. We will argue that the performance with simple tabulation is within constant factors of the performance with a truly random function, both in the regime $\varepsilon \ll 1$ (high fill) and $\alpha \ll 1$ (low fill). In fact, our proof is a combinatorial reduction to Chernoff bounds. It proves that any hash function which guarantees Chernoff bounds as in Theorem 1 achieves the ideal performance for linear probing.

Assume a fixed set of n keys have been placed into the table. We view the table as a set of runs (intervals) of consecutive keys, separated by empty slots. The running time of any operation on key q (inserting, delete, query) is bounded by the length of the run I that contains $h(q)$. Our goal is to understand the distribution of $|I|$ and bound its expectation.

For any interval I, we expect $(1 - \varepsilon)|I|$ keys to hash inside I (in the sense that $h(x) \in I$, not that x is placed by linear probing inside I). We define the *excess* of I as $\#\{x \in S \mid h(x) \in I\} - (1 - \varepsilon)|I|$. A necessary (but not sufficient) condition for I to be a run is that its excess be $\varepsilon|I|$.

Small fill. We first give a simple analysis in the case that the fill $\alpha = 1 - \varepsilon$ is bounded away from 1. Consider the dyadic intervals over $[m]$; we say an interval is on level i if it has length 2^i. An arbirary

interval I is said to have *height* i if it contains a dyadic interval on level i, but none on level $i+1$. If I has height i, its length is $|I| \in [2^i, 2^{i+2}]$.

Now let I be the run around $h(q)$. We wish to bound the probability that I has height i. We give the following necessary condition:

Claim 20. *Assume I has height i. Let j be maximal such that $2^j < \varepsilon \cdot 2^{i-2}$. Consider the dyadic interval of level j containing $h(q)$, and $2^{i+2-j} - 1$ intervals of level j on both sides. Then one of these intervals contains more than $(1 - \frac{\varepsilon}{2})2^j$ keys.*

Proof. Assume all relevant level-j intervals have excess at most $\frac{\varepsilon}{2}2^j$. The run I has length $|I| \le 2^{i+2}$, so it is covered by the union of these intervals. The run I contains some number $c \ge 2^{i-j} - 2$ of these intervals, and properly intersects at most two. The excess of I must be at least $\varepsilon|I| \ge c \cdot \varepsilon 2^j$. Each interval contained in I can contribute at most $\frac{\varepsilon}{2}2^j$ to the excess, so it remains to account for $c\varepsilon \cdot 2^{j-1} \ge (2^{i-j}-2)\varepsilon \cdot 2^{j-1} \ge \varepsilon \cdot 2^{i-2}$ keys in the excess of I. These can only come from the intervals that are properly intersected. Those could contribute all their keys to the excess (e.g. all keys are on the last/first position of the interval). However, they have at most $(1 - \frac{\varepsilon}{2})2^j$ keys, which is insufficient by choice of j. □

Crucially, the intervals considered in the claim are independent of the positioning of I around $h(q)$. Thus, we can bound the probability that I has level i by the probability that one of these $2^{i+3-j} \le 32/\varepsilon = O(1)$ intervals has too many elements. We use the Poisson approximation to Chernoff bounds from (3) to analyze an interval of level j, with $\mu = (1-\varepsilon) \cdot 2^j$. By a union bound over all intervals:

$$\Pr[I \text{ has height } i] \;\le\; O(1) \cdot \left(\frac{(1-\varepsilon/2)2^j}{(1-\varepsilon)2^j}\right)^{\Omega(\mu)} + n^{-\gamma} \;\le\; \alpha^{\Omega(\alpha 2^j)} + n^{-\gamma} \;\le\; \alpha^{\Omega(\alpha 2^i)} + n^{-\gamma}$$

Since $|I| = \Theta(2^i)$, we have shown that $\Pr[|I| = x] \le \alpha^{\Omega(\alpha x)}$. For constant fill, we get the exponential decay $\Pr[|I| = x] \le 2^{-\Omega(x)}$.

Large fill. We now consider the case $\varepsilon = o(1)$. We will get an expected time of $O(1/\varepsilon^2)$, matching within a constant factor the bound of Knuth [**?**] for truly random hash functions. For 5-independent hashing, Pagh et al. [**?**] showed a weaker bound of $O(1/\varepsilon^{13/6})$.

Let I be the run around $h(q)$ and i its height. We know that I has excess $\varepsilon|I| \ge \varepsilon 2^i$. We say that a dyadic interval on level j is "dangerous for level i" if it has excess at least:

$$\Delta_{j,i} \;=\; \tfrac{\varepsilon \cdot 2^i}{21}/2^{(i-j)/4} \;=\; \tfrac{\varepsilon}{21} \cdot 2^{\frac{3}{4}i + \frac{1}{4}j}.$$

We can decompose I into maximal dyadic intervals: at most two dyadic intervals on each level $j = 0 \ldots i$. For technical reasons, this decomposition is "rounded to level j_0", where j_0 is the smallest possible value satisfying $\Delta_{j_0,i} \le 2^{j_0}$. Formally, the decomposition rounded to level j_0 is obtained from the dyadic decomposition by discarding all intervals at level below j_0, and including one interval of level j_0 on both sides (each spanning the discarded smaller intervals).

Claim 21. *If I is a run of height i, one of the dyadic intervals in its rounded decomposition is dangerous for level i.*

Proof. Assume that no interval is dangerous for level i. We sum their excesses and show that I cannot have the required excess. Any keys in the two rounding intervals of level j_0 can potentially be in I (all keys in such intervals can hash to the beginning/end). If these intervals are no dangerous, they could contribute

$\Delta_{j_0,i}+(1-\varepsilon)2^{j_0} < 2^{j_0+1}$ keys to the excess of I. An interval on level $j \geq j_0$ from the proper decomposition contributes at most $\Delta_{j,i}$ to the excess of I, and there are at most 2 such intervals for all j. Thus, the total excess is:

$$2^{j_0+2} + 2\sum_{j\geq j_0} \Delta_{j,i} = 2^{j_0+2} + 2\sum_{j\geq j_0} \frac{\varepsilon 2^i}{21}/2^{(i-j)/4}$$

We have $\sum_{j\leq i} 1/2^{(i-j)/4} \leq \frac{1}{1-2^{-1/4}} \leq 6.3$. By choice of j_0, $2^{j_0} \leq 2\Delta_{j_0,i} \leq \frac{2\varepsilon 2^i}{21}$. Thus, the total excess is at most $(\frac{8}{21} + \frac{12.6}{21})\varepsilon \cdot 2^i < \varepsilon \cdot 2^i$, contradiction. □

We now bound the probability that a fixed interval of level j is dangerous for level i, based on the normal approximation to Chernoff bounds, from (2). We have mean $\mu = (1-\varepsilon)2^j$ and deviation $\delta\mu = \Delta_{i,j} = \Theta(\varepsilon 2^{\frac{3}{4}i+\frac{1}{4}j})$. Therefore $\delta = \Delta_{i,j}/\mu = \Theta(\varepsilon 2^{\frac{3}{4}(i-j)})$. Note that $\delta < 1$ by choice of j_0. Hence, for any constant γ:

$$\Pr[\text{level } j \text{ node dangerous for } i] \leq e^{-\Omega(\delta^2\mu)} + n^{-\gamma} \leq e^{-\Omega((\varepsilon\cdot 2^{3(i-j)/4})^2 2^j)} + n^{-\gamma} = e^{-\Omega(\varepsilon^2 2^{i+(i-j)/2})} + n^{-\gamma}.$$

We now bound the probability that $h(q)$ is contained in a run of height i. Consider the dyadic interval of level $i+1$ above $h(q)$, as well as the intervals before and after (at the same level). Any run of height i containing $h(q)$ must be contained in the union of these three intervals. Thus, a necessary condition for $h(q)$ to be in a run of height i is that the union of the three intervals above $h(q)$ contain some dyadic subinterval that is dangerous for level i. We analyze this by union bound. There are exactly $3 \cdot 2^{i+1-j}$ subintervals on level j. Hence the probability is at most:

$$n^{-\gamma} + \sum_j 3 \cdot 2^{i+1-j} \cdot e^{-\Omega(\varepsilon^2 2^{i+(i-j)/2})} = n^{-\gamma} + e^{-\Omega(\varepsilon^2 2^i)} \sum_j 6 \cdot 2^{i-j}/e^{\Omega(\varepsilon^2 2^{(i-j)/2})}$$

This sum decreases exponentially with $i-j$, so it is dominated by $j = i$. Thus, the probability of a run of height i is at most $e^{-\Omega(\varepsilon^2 2^i)} + n^{-\gamma}$. If the run I has height i, its length is $|I| \in [2^i, 2^{i+2}]$.

We have shown that $|I|$ has a geometric distribution: $\Pr[|I| = x] \leq e^{-\Omega(\varepsilon^2 x)} + n^{-\gamma}$. This has expectation $O(1/\varepsilon^2)$. Furthermore, its variance is $O(1/\varepsilon^4)$, and any bounded moment is polynomial in $1/\varepsilon$.

6 Fourth Moment Bounds

Consider distributing a set S of n balls into m bins truly randomly. For the sake of generality, let each element have a weight of w_i. We designate a query ball $q \notin S$, and let W be the total weight of the elements landing in bin $F(h(q))$, where F is an arbitrary function. With $\mu = \mathbf{E}[W] = \frac{1}{m}\sum w_i$, we are interested in the 4th moment of the bin size: $\mathbf{E}[(W-\mu)^4]$.

Let X_i be the indicator that ball $i \in S$ lands in bin $F(h(q))$, and let $Y_i = X_i - \frac{1}{m}$. We can rewrite $W - \mu = \sum_i Y_i w_i$, so:

$$\mathbf{E}[(W-\mu)^4] = \sum_{i,j,k,l\in S} w_i w_j w_k w_l \cdot \mathbf{E}[Y_i Y_j Y_k Y_l]. \tag{7}$$

The terms in which some element appears exactly once are zero. Indeed, if $i \notin \{j,k,l\}$, then $\mathbf{E}[Y_i Y_j Y_k Y_l] = \mathbf{E}[Y_i] \cdot \mathbf{E}[Y_j Y_k Y_l]$, which is zero since $\mathbf{E}[Y_i] = 0$. Thus, the only nonzero terms arise from:

- four copies of one element ($i = j = k = l$), giving the term $\left(\frac{1}{m} \pm O(\frac{1}{m^2})\right) w_i^4$.

- two distinct elements $s \neq t$, each appearing twice. There are $\binom{4}{2} = 6$ terms for each s, t pair, and each term is $O(\frac{1}{m^2})w_s^2 w_t^2$.

This gives the standard 4th moment bound:

$$\mathbf{E}\big[(W-\mu)^4\big] \;=\; \frac{1}{m}\sum_i w_i^4 + \frac{O(1)}{m^2}\big(\sum_i w_i^2\big)^2. \tag{8}$$

This bound holds even if balls are distributed by 5-independent hashing: the balls in any 4-tuple hit the bin chosen by $h(q)$ independently at random. On the other hand, with 4-independent hashing, this bound can fail quite badly [**?**].

If the distribution of balls into bins is achieved by simple tabulation, we will show a slightly weaker version of (8):

$$\mathbf{E}\big[(W-\mu)^4\big] \;=\; \frac{1}{m}\sum_i w_i^4 + O\left(\frac{1}{m^2} + \frac{4^c}{m^3}\right)\cdot\Big(\sum_i w_i^2\Big)^2. \tag{9}$$

In §6.1, we show how to analyze the 4th moment of a fixed bin (which requires 4-independence by standard techniques). Our proof is a combinatorial reduction to Cauchy–Schwarz. In §6.2, we let the bin depend on the hash code $h(q)$. This requires 5-independence by standard techniques. To handle tabulation hashing, §6.3 shows a surprising result: among any 5 keys, at least one hashes independently of the rest.

We note that the bound on the 4th moment of a fixed bin has been indendently discovered by [**?**] in a different context. However, that work is not concerned with a query-dependent bin, which is the most surprising part of our proof.

6.1 Fourth Moment of a Fixed Bin

We now attempt to bound the terms of (7) in the case of simple tabulation. Since simple tabulation is 3-independent [**?**], any terms that involve only 3 distinct keys (i.e. $|\{i,j,k,l\}| \leq 3$) have the same expected value as established above. Thus, we can bound:

$$\mathbf{E}[(W-\mu)^4] \;=\; \frac{1}{m}\sum_i w_i^4 + \frac{O(1)}{m^2}\big(\sum_i w_i^2\big)^2 \;+\; \sum_{i\neq j\neq k\neq l} w_i w_j w_k w_l \cdot \mathbf{E}[Y_i Y_j Y_k Y_l].$$

Unlike the case of 4-independence, the contribution from distinct i, j, k, l will not be zero. We begin with the following simple bound on each term:

Claim 22. *For distinct i, j, k, l, $\mathbf{E}[Y_i Y_j Y_k Y_l] \;=\; O(\frac{1}{m^3})$.*

Proof. We are looking at the expectation of $Z = (X_i - \frac{1}{m})(X_j - \frac{1}{m})(X_k - \frac{1}{m})(X_l - \frac{1}{m})$. Note that Z is only positive when an *even* number of the four X's are 1:

1. the case $X_i = X_j = X_k = X_l = 1$ only happens with probability $\frac{1}{m^3}$ by 3-independence. The contribution to Z is $(1-\frac{1}{m})^4 < 1$.
2. the case of two 1's and two 0's happens with probability at most $\binom{4}{2}\frac{1}{m^2}$, and contributes $\frac{1}{m^2}(1-\frac{1}{m})^2 < \frac{1}{m^2}$ to Z.
3. the case of $X_i = X_j = X_k = X_l = 0$ contributes $\frac{1}{m^4}$ to Z.

Thus, the first case dominates and $\mathbf{E}[Z] = O(\frac{1}{m^3})$. □

If one of $\{i,j,k,l\}$ contains a unique position-character, its hash code is independent of the other three. In this case, the term is zero, as the independent key factors out of the expectation and $\mathbf{E}[Y_i] = 0$. We are left with analyzing 4-tuples with no unique position-characters; let $A \subseteq S^4$ contain all such 4-tuples. Then:

$$\sum_{i\neq j\neq k\neq l} w_i w_j w_k w_l \cdot \mathbf{E}[Y_i Y_j Y_k Y_l] \;=\; O\left(\tfrac{1}{m^3}\right) \cdot \sum_{(i,j,k,l)\in A} w_i w_j w_k w_l.$$

Imagine representing a tuple from A as a $4 \times q$ matrix, with every key represented in a row. There are four types of columns that we may see: columns that contain a single character in all rows (type 1), and columns that contain two distinct characters, each appearing in two rows (type $j \in \{2,3,4\}$ means that row j has the same character as row 1). According to this classification, there are 4^q possible matrix types.

Claim 23. *Fix a fixed matrix type, and let $B \subseteq A$ contain all tuples conforming to this type. Then, $\sum_{(i,j,k,l)\in B} w_i w_j w_k w_l \leq \left(\sum_i w_i^2\right)^2$.*

Proof. We first group keys according to their projection on the type-1 characters. We obtain a partition of the keys $S = S_1 \cup S_2 \cup \cdots$ such that S_t contains keys that are identical in the type-1 coordinates. Tuples that conform to the fixed matrix type, $(i,j,k,l) \in B$, must consist of four keys from the same set, i.e. $i,j,k,l \in S_t$. Below, we analyze each S_t separately and bound the tuples from $(S_t)^4$ by $\left(\sum_{i\in S_t} w_i^2\right)^2$. This implies the lemma by convexity, as $\sum_t \left(\sum_{i\in S_t} w_i^2\right)^2 \;\leq\; \left(\sum_i w_i^2\right)^2$.

For the remainder, fix some S_t. If $|S_t| < 4$, there is nothing to prove. Otherwise, there must exist at least one character of type different from 1, differentiating the keys. By permuting the set $\{i,j,k,l\}$, we may assume a type-2 character exists. Group keys according to their projection on all type-2 characters. We obtain a partition of the keys $S_t = T_1 \cup T_2 \cup \cdots$ such that T_a contains keys that are identical in the type-2 coordinates.

A type-conforming tuple $(i,j,k,l) \in B$ must satisfy $i,j \in T_a$ and $k,l \in T_b$ for $a \neq b$. We claim a stronger property: for any $i,j \in T_a$ and every $b \neq a$, there exists at most one pair $k,l \in T_b$ completing a valid tuple $(i,j,k,l) \in B$. Indeed, for type-1 coordinates, k and l must be identical to i on that coordinate. For type 3 and 4 coordinates, k and l must reuse the characters from i and j ($k \leftarrow i, l \leftarrow j$ for type 3; $k \leftarrow j, l \leftarrow i$ for type 4).

Let $X \subset (T_a)^2$ contain the pairs $i,j \in T_a$ which can be completed by one pair $k,l \in T_b$. Let $Y \subset (T_b)^2$ contain the pairs $k,l \in T_b$ which can be completed by $i,j \in T_a$. There is a bijection between X and Y; let it be $f : X \mapsto Y$. We can now apply the Cauchy-Schwarz inequality:

$$\begin{aligned}\sum_{(i,j,k,l)\in B\cap(T_a\times T_b)} w_i w_j w_k w_l \;&=\; \sum_{(i,j)\in X,\ (k,l)=f(i,j)} (w_i w_j)\cdot(w_k w_l) \\ &\leq\; \sqrt{\left(\sum_{(i,j)\in X} (w_i w_j)^2\right)\left(\sum_{(k,l)\in Y} (w_k w_l)^2\right)}\end{aligned}$$

But $\sum_{(i,j)\in X} w_i^2 w_j^2 \leq \left(\sum_{i\in T_a} w_i^2\right)^2$. Thus, the equation is further bounded by $\left(\sum_{i\in T_a} w_i^2\right)\left(\sum_{k\in T_b} w_k^2\right)$.

Summing up over all T_a and T_b, we obtain:

$$\sum_{(i,j,k,l)\in B\cap(S_t)^4} w_i w_j w_k w_l \;\leq\; \sum_{a,b}\left(\sum_{i\in T_a} w_i^2\right)\left(\sum_{k\in T_b} w_k^2\right) \;\leq\; \left(\sum_{i\in S_t} w_i^2\right)^2$$

This completes the proof of the claim. □

The bound of Claim 23 is multiplied by 4^q, the number of matrix types. We have thus shown (9).

6.2 Fourth Moment of a Query-Dependent Bin

We now aim to bound the 4th moment of a bin chosen as a function F of $h(q)$, where q is a designated query ball. This requires dealing with 5 keys (i, j, k, l and the query q). Even though simple tabulation is only 3-independent, we will prove the following intriguing independence guarantee in §6.3:

Theorem 24. *With simple tabulation, in any fixed set of 5 distinct keys, there is a key whose hash is independent of the other 4 hash codes.*

As a side note, we observe that this theorem essentially implies that *any* 4-independent tabulation based scheme is also 5-independent. In particular, this immediately shows the 5-independence of the scheme from [**?**] (which augments simple tabulation with some derived characters). This fact was already known [**?**], albeit with a more complicated proof.

In the remainder of this section, we use Theorem 24 to derive the 4th moment bound (9). As before, we want to bound terms $w_i w_j w_k w_l \cdot \mathbf{E}[Y_i Y_j Y_k Y_l]$ for all possible configurations of (i, j, k, l). Remember that $q \notin S$, so $q \notin \{i, j, k, l\}$. These terms can fall in one of the following cases:

- All keys are distinct, and q hashes independently. Then, the contribution of i, j, k, l to bin $F(h(q))$ bin is the same as to any fixed bin.
- All keys are distinct, and q is dependent. Then, at least one of $\{i, j, k, l\}$ must be independent of the rest and q; say it is i. But then we can factor i out of the product: $\mathbf{E}[Y_i Y_j Y_k Y_l] = \mathbf{E}[Y_i] \cdot \mathbf{E}[Y_j Y_k Y_l]$. The term is thus zero, since $\mathbf{E}[Y_i] = 0$.
- Three distinct keys, $\big|\{i, j, k, l\}\big| = 3$. This case is analyzed below.
- One or two distinct keys: $\big|\{i, j, k, l\}\big| \leq 2$. By 3-independence of simple tabulation, all hash codes are independent, so the contribution of this term is the same as in the case of a fixed bin.

To summarize, the 4th moment of bin $F(h(q))$ is the same as the 4th moment of a fixed bin, plus an additional term due to the case $\big|\{i, j, k, l\}\big| = 3$. The remaining challenge is to understand terms of the form $w_i^2 w_j w_k \mathbf{E}\big[Y_i^2 Y_j Y_k\big]$. We first prove the following, which is similar to Claim 22:

Claim 25. *For distinct i, j, k, $\mathbf{E}[Y_i^2 Y_j Y_k] = O(\frac{1}{m^2})$.*

Proof. By 3-independence of simple tabulation, Y_i and Y_j are independent (these involve looking at the hashes of i, j, q). For an upper bound, we can ignore all outcomes $Y_i^2 Y_j Y_k < 0$, i.e. when Y_j and Y_k have different signs. On the one hand, $Y_j = Y_k = 1 - \frac{1}{m}$ with probability $O(\frac{1}{m^2})$. On the other hand, if $Y_j = Y_k = -\frac{1}{m}$, the contribution to the expectation is $O(\frac{1}{m^2})$. □

Assume $w_j \geq w_k$ by symmetry. If k hashes independently of $\{i, j, q\}$, the term is zero, since $\mathbf{E}[Y_k] = 0$ can be factored out. Otherwise, the term contributes $O(w_i^2 w_j^2 / m^2)$ to the sum.

Claim 26. *For any distinct i, j, q, there is a unique key k such that $h(k)$ depends on $h(i), h(j), h(k)$.*

Proof. We claim that if any of $\{i, j, k, q\}$ has a unique position-character, all keys are independent. Indeed, the key with a unique position-character is independent of the rest, which are independent among themselves by 3-independence.

Thus, any set $\{i, j, q\}$ that allows for a dependent k cannot have 3 distinct position-characters on one position. In any position where i, j, and q conincide, k must also share that position-character. If i, j, and q contain two distinct distinct characters on some positon, k must contain the one that appears once. This determines k. □

For any i and j, we see exactly one set $\{i, j, k\}$ that leads to bad tuples. By an infinitesimal perturbation of the weights, each such set leads to $\binom{4}{2} = 6$ tuples: we have to choose two positions for i, and then j is the remaining key with larger weight. Thus, the total contribution of all terms (i, j, k, l) with 3 distinct keys is $O(\sum_{i,j} \frac{w_i^2 w_j^2}{m^2}) = O(\frac{1}{m^2})(\sum_i w_i^2)^2$. This completes the proof of (9).

6.3 Independence Among Five Keys

The section is dedicated to proving Theorem 24. We first observe the following immediate fact:

Fact 27. *If, restricting to a subset of the characters (matrix columns), a key $x \in X$ hashes independently from $X \setminus \{x\}$, then it also hashes independently when considering all characters.*

If some key contains a unique character, we are done by peeling. Otherwise, each column contains either a single value in all five rows, or two distinct values: one appearing in two rows, and one in three rows. By Fact 27, we may ignore the columns containing a single value. For the columns containing two values, relabel the value appearing three times with 0, and the one appearing twice with 1. By Fact 27 again, we may discard any duplicate column, leaving at most $\binom{5}{2}$ distinct columns.

Since the columns have weight 2, the Hamming distance between two columns is either 2 or 4.

Lemma 28. *If two columns have Hamming distance 4, one hash value is independent.*

Proof. By Fact 27, we ignore all other columns. Up to reordering of the rows, the matrix is:

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

By 3-independence of character hashing, keys 1, 3, and 5 are independent. But keys 2 and 4 are identical to keys 1 and 3. Thus, key 5 is independent from the rest. □

We are left with the case where all column pairs have Hamming distance 2. By reordering of the rows, the two columns look like the matrix in (a) below. Then, there exist only two column vectors that are at distance two from both of the columns in (a):

$$\text{(a)} \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \text{(b)} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \qquad \text{(c)} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

If the matrix does not contain column (b), then keys 4 and 5 are identical, a contradiction. Thus, the matrix must contain columns (a) and (b), with (c) being optional. If (c) appears, discard it by Fact 27. We are left with the matrix:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Now, observe that the hash code of row 1 is just the xor of the hash codes of rows 2–4, $h(1) = h(2) \oplus h_C(3) \oplus h_C(4)$. Indeed, the codes of the one characters in each column cancel out, leaving us with an xor of the zeros in each column. We claim that row 5 is independent of rows 2–4. This immediately implies that row 5 is independent of all others, since row 1 is just a function of rows 2–4.

Independence of row 5 from rows 2–4 follows by peeling. Each of rows 2, 3, and 4 have a position character not present in 5, so they are independent of 5. This completes the proof of Theorem 24.

A Experimental evaluation

In this section, we make some simple experiments comparing simple tabulation with other hashing schemes, both on their own, and in applications. Most of our experiments are the same as those in [**?**] except that we here include simple tabulation whose relevance was not realized in [**?**]. We will also consider Cuckoo hashing which was not considered in [**?**].

Recall the basic message of our paper that simple tabulation in applications shares many of the strong mathematical properties normally associated with an independence of at least 5. For example, when used in linear probing, the expected number of probes is constant for any set of input keys. With sufficiently random input, this expected constant is obtained by any universal hashing scheme [**?**], but other simple schemes fail on simple structured inputs like dense intervals or arithmetic progressions, which could easily occur in practice [**?**].

Our experiments consider two issues:

- How fast is simple tabulation compared with other realistic hashing schemes on random input? In this case, the quality of the hash function doesn't matter, and we are only comparing their speed.
- What happens to the quality on structured input. We consider the case of dense intervals, and also the hypercube which we believe should be the worst input for simple tabulation since it involves the least amount of randomness.

We will now briefly review the hashing schemes considered in our experiments. The focus will be on the common cases of 32 and 64 bit keys. If the initial keys are much bigger, we can typically first apply universal hashing to get down to a smaller domain, e.g., collision free down to a domain of size n^2. To achieve expected $O(1)$ time for linear probing, it suffices to map universally to a domain of just $O(n)$ [**?**].

A.1 Multiplication-shift Hashing

The fastest known hashing schemes are based on a multiplication followed by a shift.

Univ-mult-shift. If we are satisfied with plain universal hashing, then as shown in [**?**], we pick a random odd number a from the same ℓ-bit domain as the keys. If the desired output is ℓ_{out}-bit keys, we compute the universal hash function:

$$h_a(x) = (a \star x) >> (\ell - \ell_{out}).$$

This expression should be interpreted according to the C programming language. In particular, $\star$ denotes standard computer multiplication where the result is truncated to the same size as that of its largest operand. Here this means multiplication modulo 2^ℓ. Also, >> is a right shift taking out least significant bits. Mathematically, this is integer division by $2^{\ell-\ell_{out}}$. Note that this scheme is far from 2-independent, e.g., if two keys differ in only their least significant bit, then so does their hash values. However, the scheme is universal which suffices, say, for expected constant times in chaining.

2-indep-mult-shift. For 2-independent hashing, we use the scheme from [**?**]. We pick a random 2ℓ-bit multiplier a (which does not need to be odd), and a 2ℓ bit number b. Now we compute:

$$h_{a,b}(x) = (a \star x + b) >> (2\ell - \ell_{out}).$$

This works fine with a single 64-bit multiplication when $\ell = 32$. For $\ell = 64$, we would need to simulate 128-bit multiplication. In this case, we have a faster alternative used for string hashing [**?**], viewing the key

x as consisting of two 32-bit keys x_1 and x_2. For a 2-independent 32-bit output, we pick three random 64-bit numbers a_1 and a_2 and b, and compute

$$h_{a_1,a_2,b}(x_1x_2) = ((a_1+x_2)*(a_2+x_1)+b)>>32.$$

Concatenating two such values, we get a 64-bit 2-independent hash value using just two 64-bit multiplications.

A.2 Polynomial Hashing

For general k-independent hashing, we have the classic implementation of Carter and Wegman [**?**] by a degree $k-1$ polynomial over some prime field:

$$h(x) = \left(\sum_{i=0}^{k-1} a_i x^i \bmod p\right) \bmod 2^{\ell_{out}} \tag{10}$$

for some prime $p \gg 2^{\ell_{out}}$ with each a_i picked randomly from $[p]$. If p is an arbitrary prime, this method is fairly slow because '$\bmod p$' is slow. However, Carter and Wegman [**?**] pointed out that we can get a fast implementation using shifts and bitwise Boolean operations if p is a so-called Mersenne prime of the form $2^i - 1$.

5-indep-Mersenne-prime. We use the above scheme for 5-independent hashing. For 32-bit keys, we use $p = 2^{61} - 1$, and for 64-bit keys, we use $p = 2^{89} - 1$.

For the practical implementation, recall that standard 64-bit multiplication on computers discards overflow beyond the 64 bits. For example, this implies that we may need four 64-bit multiplications just to implement a full multiplication of two numbers from $[2^{61} - 1]$. This is why specialized 2-independent schemes are much faster. Unfortunately, we do not know a practical generalization for higher independence.

A.3 Tabulation-Based Hashing

The basic idea in tabulation based schemes is to replace multiplications with lookups in tables that are small enough to fit in fast memory.

simple-table. Simple tabulation is the basic example of tabulation based hashing. A key $x = x_1 \cdots x_c$ is divided into c characters. For $i = 1..c$, we have a table T_i providing a random value $T_i[x_i]$ with a random value, and then we just return the xor of all the $T_i[x_i]$. Since the tables are small, it is easy to populate them with random data (e.g. based on atmospheric noise `http://random.org`). Simple tabulation is only 3-independent.

We are free to chose the size of the character domain, e.g., we could use 16-bit characters instead of 8-bit characters, but then the tables would not fit in the fast L1 cache. The experiments from [**?**] indicate that 8-bit characters give much better performance, and that is what we use here.

5-indep-TZ-table. To get higher independence, we can compute some additional "derived characters" and use them to index into new tables, like the regular characters. Thorup and Zhang [**?**, **?**] presented a fast such scheme for 5-independet hashing. With $c = 2$ characters, they simply use the derived character $x_1 + x_2$. For $c > 2$, this generalizes with $c - 1$ derived characters and a total of $2c - 1$ lookups for 5-independent hashing. The scheme is is rather complicated to implement, but runs well.

Hashing random keys		32-bit computer	64-bit computer
bits	hashing scheme	hashing time (ns)	
32	univ-mult-shift	1.87	2.33
32	2-indep-mult-shift	5.78	2.88
32	5-indep-Mersenne-prime	99.70	45.06
32	5-indep-TZ-table	10.12	12.66
32	simple-table	4.98	4.61
64	univ-mult-shift	7.05	3.14
64	2-indep-mult-shift	22.91	5.90
64	5-indep-Mersenne-prime	241.99	68.67
64	5-indep-TZ-table	75.81	59.84
64	simple-table	15.54	11.40

Table 1: Average time per hash computation for 10 million hash computations.

A.4 Hashing in Isolation

Our first goal is to time the different hashing schemes when run in isolation. We want to know how simple tabulation compares in speed to the fast multiplication-shift schemes and to the 5-independent schemes whose qualities it shares. We compile and run the same C code on two different computers:

32-bit computer: Single-core Intel Xeon 3.2 GHz 32-bit processor with 2048KB cache, 32-bit addresses and libraries.

64-bit computer: Dual-core Intel Xeon 2.6 GHz 64-bit processor with 4096KB cache, 64-bit addresses and libraries.

Table 1 presents the timings for the different hashing schemes, first mapping 32-bit keys to 32-bit values, second mapping 64-bit keys to 64-bit values. Not surprisingly, we see that the 64-bit computer benefits more than the 32-bit computer when 64-bit multiplications is critical; namely in univ-mult-shift for 64 bits, 2-indep-mult-shift, and 5-indep-Mersenne-prime.

As mentioned, the essential difference between our experiments and those in [**?**] is that simple tabulation is included, and our interest here is how it performs relative to the other schemes. In the case of 32-bits keys, we see that in both computers, the performance of simple tabulation is similar to 2-indep-mult-shift. Also, not surprisingly, we see that it is more than twice as fast as the much more complicated 5-indep-TZ-table.

When we go to 64-bits, it may be a bit surprising that simple tabulation becomes more than twice as slow, for we do exactly twice as many look-ups. However, the space is quadrupled with twice as many tables, each with twice as big entries, moving up from 1KB to 8KB, so the number of cache misses may increase.

Comparing simple tabulation with the 2-indep-mult-shift, we see that it is faster on the 32-bit computer and less than twice as slow on the 64-bit computer. We thus view it as competitive in speed.

The competetiveness of simple tabulation compared with multiplication-shift based methods agrees with the experiments of Thorup [**?**] from more than 10 years ago on older computer architectures. The experiments from [**?**] did not include schemes of higher independence.

Linear probing with random keys	32-bit computer	64-bit computer
hashing scheme	update time (nanoseconds)	
univ-mult-shift	141	149
2-indep-mult-shift	151	157
5-indep-Mersenne-prime	289	245
5-indep-TZ-table	177	211
simple-table	149	166

Table 2: Linear probing with random 32-bit keys. The time is averaged over 10 million updates to set with 1 million keys in linear probing table with 2 million entries.

A.5 Linear Probing

We now consider what happens when we use the different hashing schemes with linear probing. In this case, the hash function needs good random properties are need for good performance on worst-case input. We consider 2^{20} 32-bit keys in a table with 2^{21} entries. The table therefore uses 8MB space, which does not fit in the cache of either computer, so there will be competition for the cache. Each experiment averaged the update time over 10 million insert/delete cycles. For each input type, we ran 100 such experiments on the same input but with different random seeds for the hash functions.

Random input. First we consider the case of a random input, where the randomization properties of the hash function are irrelevant. This means that the focus is on speed just like when we ran the experiments in isolation. Essentially, the cost per update should be that of computing the hash value, as in Table 1, plus a common additive cost: a random access to look up the hash entry plus a number of sequential probes. The average number of probes per update was tightly concentrated around 3.28 for all schemes, deviating by less than 0.02 over the 100 experiments.

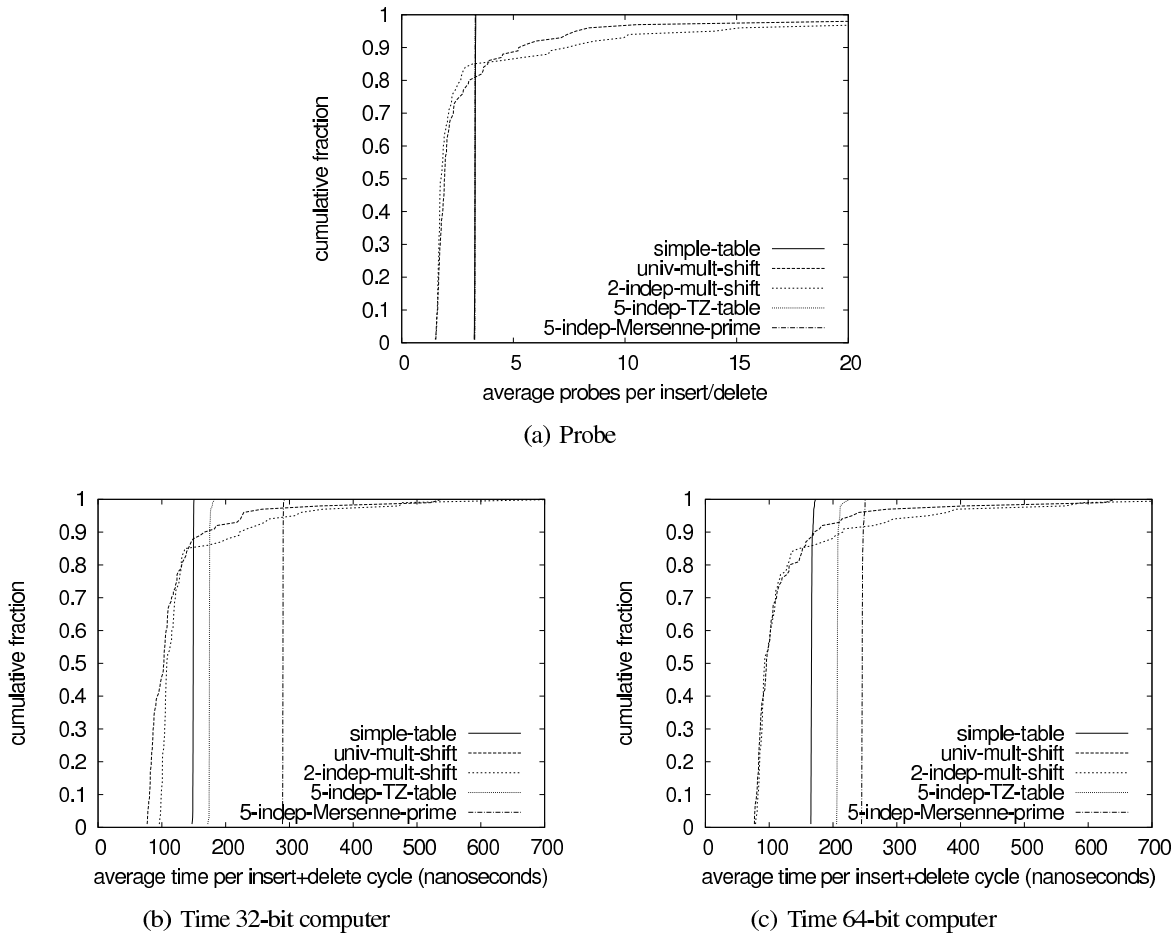
An interesting new issue is that the different schemes schemes now have to compete with the linear probing table for the cache. In particular, this could hurt the tabulation based schemes. Another issue is that when schemes are integrated with an application, the optimizing compiler may have many more opportunities for pipelining etc. The results for random input are presented in Table 2. Within the 100 experiments, the deviation for each data point was less than 1%, and here we just present the mean.

Compared with Table 1, we see that our 32-bit computer performs better than the 64-bit computer on linear probing. In Table 1 we had that the 64-bit processor was twice as fast at the hash computations based on 64-bit multiplication, but in Table 2, when combined with linear probing, we see that it is only faster in the most extreme case of 5-indep-Mersenne-prime. One of the more surprising outcomes is that 5-indep-Mersenne-prime is so slow compared with the tabulation based schemes on the 64-bit computer. We had expected the tabulation based schemes to take a hit from cache competition, but the effect appears to be minor.

The basic outcome is that simple tabulation in linear probing with random input is competitive with the fast multiplication-shift based scheme and about 20% faster than the fastest 5-independent scheme (which is much more complicated to implement). We note that we cannot hope for a big multiplicative gain in this case, since the cost is dominated by the common additive cost from working the linear probing table itself.

Structured input. We now consider the case where the input keys are structured in the sense of being drawn in random order from a dense interval: a commonly occurring case in practice which is known to

cause unreliable performance for most simple hashing schemes [**?**, **?**, **?**]. The results are shown in Figure **??**. For each hashing scheme, we present the average number of probes for each of the 100 experiments as a cumulative distribution function. We see that simple tabulation and the 5-independent schemes remain tightly concentrated while the multiplication-shift schemes have significant variance, as observed also in [**?**]. This behavior is repeated in the timings on the two computers, but shifted due to difference in speed for the hash computations.



(a) Probe

(b) Time 32-bit computer

(c) Time 64-bit computer

Figure 2: Keys from dense interval. The multiplication-shift schemes sometimes use far more probes, which also shows in much longer running times.

Thus, among simple fast hashing schemes, simple tabulation stands out in not failing on a dense interval. Of course, it might be that simple tabulation had a different worst-case input. A plausible guess is that the worst instance of simple tabulation is the hypercube, which minimizes the amount of random table entries used. In our case, for 2^{20} keys, we experimented with the set $[32]^4$, i.e., we only use 32 values for each of the 4 characters. The results for the number of probes are presented in Figure **??**.

Thus, simple tabulation remains extremely robust and tightly concentrated, but once again the multiplication-shift schemes fail (this time more often but less badly). The theoretical explanation from [**?**] is that multiplication-shift fails on arithmetic sequences, and in the hypercube we have many different but shorter arithmetic sequences. It should be said that although it cannot be seen in the plots, the structured in-

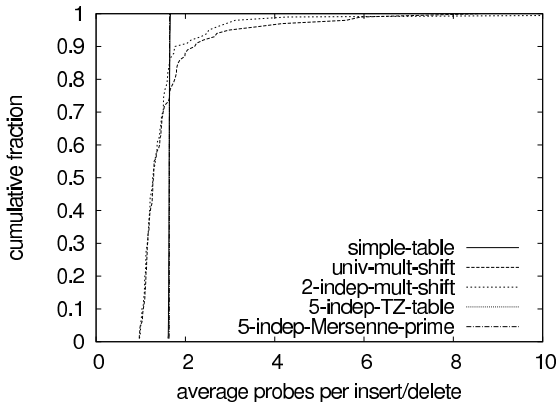


Figure 3: Keys from hyper cube.

puts did lead to more deviation in probes for simple tabulation: the deviation from 3.28 over 100 independent runs grew from below 0.5% with random input to almost 1% with any of the structured input.

Obviously, no experiment can confirm that simple tabulation is robust for all possible inputs. Our theoretical analysis implies strong concentration, e.g., in the sense of constant variance, yet the hidden constants are large. Our experiments suggest that the true constants are very reasonable.

Cuckoo hashing. Our results show that the failure probability in constructing a cuckoo hashing table is $O(n^{-1/3})$. A pertinent question is whether the constants hidden by the O-notation are too high from a practical point of view. Experiments cannot conclusively prove that this constant is always small, since we do not know the worst instance. However, as for linear probing, a plausible guess that the instance eliciting the worst behavior is a hypercube: $S = A^c$, for $A \subset \Sigma$. We made 10^5 independent runs with the following input instances:

32-bit keys: Tabulation uses $c = 4$ characters. We set $A = [32]$, giving $32^4 = 2^{20}$ keys in S. The empirical success probability was 99.4%.

64-bit keys: Tabulation uses $c = 8$ characters. We set $A = 8$, giving $8^8 = 2^{24}$ keys in S. The empirical success probability was 97.1%.

These experiments justify the belief that our scheme is effective in practice.

It has already been shown conclusively that weaker multiplication schemes do not perform well. Dietzfelbinger and Schellbach [**?**] show analytically that, when S is chosen uniformly at random from the universe $[n^{12/11}]$ or smaller, cuckoo hashing with 2-independent multiplicative hashing fails with probability $1 - o(1)$. This is borne out in the experiments of [**?**], which give failure probability close to 1 for random sets that are dense in the universe. On the other hand, the more complicated tabulation hashing of Thorup and Zhang [**?**] will perform at least as well as simple tabulation (that algorithm is a superset of simple tabulation).

A notable competitor to simple tabulation is a tailor-made tabulation hash function analyzed by Dietzfelbinger and Woelfel [**?**]. This function uses two arrays of size r and four d-independent hash functions to obtain failure probability $n/r^{d/2}$.

Let us analyze the parameters needed in a practical implementation. If we want the same space as in simple tabulation, we can set $r = 2^{10}$ (this is larger than $\Sigma = 256$, because fewer tables are needed). For a nontrivial failure probability with sets of 2^{20} keys, this would require 6-independence. In principle,

the tabulation-based scheme of [**?**] can support 6-independence with $5c - 4$ tables (and lookups). This scheme has not been implemented yet, but based on combinatorial complexity is expected to be at least twice as slow as the 5-independent scheme tested in Table 1 (i.e. 4-8 times slower than simple tabulation). Alternatively, we can compute four 6-independent hash functions using two polynomials of degree 5 on 64-bit values (e.g. modulo $2^{61}-1$). Based on Table 1, this would be two orders of magnitude slower than simple tabulation. With any of the alternatives, the tailor-made scheme is much more complicated to implement.

Acknowledgement

We are grateful to Yin Zhang for considerable help with the experiments of Appendix A.

References

[BCL+10] Vladimir Braverman, Kai-Min Chung, Zhenming Liu, Michael Mitzenmacher, and Rafail Ostrovsky. AMS without 4-wise independence on product domains. In *Proc. 27th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 119–130, 2010.

[CK09] Jeffery S. Cohen and Daniel M. Kane. Bounds on the independence required for cuckoo hashing. Manuscript, 2009.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. See also STOC'77.

[DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[Die96] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. 13th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 569–580, 1996.

[DS09] Martin Dietzfelbinger and Ulf Schellbach. On risks of using cuckoo hashing with simple universal hash classes. In *Proc. 20th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 795–804, 2009.

[DW03] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *Proc. 25th ACM Symposium on Theory of Computing (STOC)*, pages 629–638, 2003.

[Ger68] Leon Gerber. An extension of Bernoulli's inequality. *American Mathematical Monthly*, 75:875–876, 1968.

[Ind01] Piotr Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001. See also SODA'99.

[Knu63] Donald E. Knuth. Notes on open addressing. Unpublished memorandum. See `http://citeseer.ist.psu.edu/knuth63notes.html`, 1963.

[KR93] Howard J. Karloff and Prabhakar Raghavan. Randomized algorithms and pseudorandom numbers. *Journal of the ACM*, 40(3):454–476, 1993.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[MV08] Michael Mitzenmacher and Salil P. Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proc. 19th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 746–755, 2008.

[PPR09] Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with constant independence. *SIAM Journal on Computing*, 39(3):1107–1120, 2009. See also STOC'07.

[PT10] Mihai Pătraşcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 715–726, 2010.

[Sie04] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing*, 33(3):505–543, 2004. See also FOCS'89.

[SSS95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995. See also SODA'93.

[Tho00] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proc. 11th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 496–497, 2000.

[Tho09] Mikkel Thorup. String hashing for linear probing. In *Proc. 20th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–664, 2009.

[TZ04] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proc. 15th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 615–624, 2004.

[TZ09] Mikkel Thorup and Yin Zhang. Tabulation based 5-universal hashing and linear probing. In *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2009.

[WC81] Mark N. Wegman and Larry Carter. New classes and applications of hash functions. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. See also FOCS'79.