The Power of Simple Tabulation Hashing

Mihai Pătraşcu  Mikkel Thorup
AT&T Labs        AT&T Labs

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

For example, Knuth gave birth to the analysis of algorithms in 1963 [Knu63] when he analysed 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 [MV08]. However, they fail badly even for very simple input structures like an interval of consecutive keys [PPR09, PT10], giving linear probing an undeserved reputation of being non-robust.

On the other hand, the approach of algorithm design 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 [WC81]. It is known that 5-independence is sufficient [PPR09] and necessary [PT10] for linear probing. Then one can use the best available implementation of 5-independent hash functions, the tabulation-based method of [TZ04].

Here we analyze simple tabulation hashing. This scheme 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 $h(x) = T_i[x_1] \oplus \cdots \oplus T_c[x_c]$. This is a well-known scheme dating back at least to Wegman and Carter [WC81]. From a practical viewpoint, 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 realistic.

In many cases, our analysis gives the first 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 table of size $m \geq (1 + \epsilon)n$, it is possible to place a ball in one of two randomly chosen locations without any collision, with probability $1 - O(\frac{\epsilon}{n})$.
- 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 $x \not\in S$, $\Pr_{h}[x < \min h(S)] = \frac{1}{|S|+1}$.

As defined by Wegman and Carter [WC81] 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 [SSS95]; 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 [PPR09] and necessary [PT10]. For cuckoo hashing and the power of two choices, $O(\lg n)$-independence suffices. At least 6-independence is needed for the former and 4-independent for the latter [CK]. While minwise independence cannot be achieved, one can achieve $\epsilon$-minwise independence with the guarantee $(\forall)x \not\in S, \Pr_{h}[x < \min h(S)] = \frac{1}{|S|+1} + \epsilon$. For this, $\Theta(\lg \frac{1}{\epsilon})$ independence is sufficient [Ind01] and necessary [PT10]. Note that the $\epsilon$ is a bound on how well set intersection can be approximated.
The canonical construction of $k$-independent hash functions is a random degree $k - 1$ polynomial in a prime field, which has small representation but $O(k)$ evaluation time. This is unappealing when $k \approx \lg n$. Siegel [Sie04] shows that a family with superconstant independence but $O(1)$ evaluation time requires $\Omega(u^{\frac{k}{c}})$ space, i.e. it requires tabulation. He also gives a solution that uses $O(u^{1/c})$ space, $e^{O(c)}$ evaluation time, and achieves $u^{\frac{k}{(1/c)^2}}$ 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 [TZ04]. 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 which is not formally $k$-independent: cuckoo hashing works with an ad hoc hash function that combines space $O(u^{1/c})$ and polynomials of degree $O(c)$ [DW03].

1 Our results

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 diverse techniques analyzing the table dependencies for very 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 a additional designated query ball, and define $X_q$ as the number of regular balls that hash into a bin that is chosen as a function of $h(q)$. Let $\mu = E[X_q] = \frac{n}{m}$. Then for any constant $\gamma$:

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

and

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

Let us contrast this to the standard Chernoff bounds [MR95]:

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

and

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

(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 $\gamma$. In addition the exponential dependence on $\mu$ is reduced by a constant, which depends on the choice of $\gamma$ and $c$.

An alternative way to understand the bound is that our tail bound depends exponentially on $e^{\mu}$, where $e^{-\delta}$ decays to subconstant as we move further out in the tail. Thus, our bounds are sufficient for any high probability guarantee. However, compared to the standard Chernoff bound, we would have to tolerate a constant-fraction 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:

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

(2)

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

$$\Pr[X > x] < (e^{x})^{\Omega(x)} n^{-\gamma}$$

(3)

By the union bound (2) implies that no bin receives more than $O(\log n)$ balls w.h.p. This is the first realistic hash function to achieve this fundamental property. Similarly, for linear probing with a half full table, we get from (3) that the longest filled interval is of length $O(\log n)$ w.h.p.
Linear probing. Analyzing linear probing with the above concentration bounds, we will show that if the table size is $m = (1 + \varepsilon)n$, then the expected operation time is $O(1/\varepsilon^2)$, which asymptotically matches the bound of Knuth [Knu63] for a truly random function. In particular, this improves on the $O(1/\varepsilon^{13/6})$ bound from [PPR09] for 5-independent hashing. We can also show that the variance is constant for constant $\varepsilon$, whereas with 5-independent hashing it was only known to be $O(\log n)$ [PPR09, TZ09]. The derivation of these bounds from the concentration bounds is comparatively straightforward and deferred to an appendix.

Minwise independence. For minwise independence, we consider a key $q \notin S$, and let $n = |S|$. We show that if we apply tabulation based hashing to $S$, then the probability that $q$ gets the smallest hash value is $\frac{1}{n} \cdot (1 \pm O(\frac{\log^2 n}{n^{1/3}})).$ Thus, the error term is vanishingly small as our sets increase. Such an error would otherwise require $\Omega(\log n)$ independence.

Cuckoo hashing. For cuckoo hashing, we obtain a rather unintuitive result: the optimal failure probability with simple tabulation is $\Theta(n^{1/3})$.

Theorem 2. 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 the query runs in constant worst-case time. 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.

Our proof involves a heavy 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 $m$ balls are placed in $O(n)$ bins in an online fashion, choosing the least loaded bin at each time, the maximum load is $O(\log \log n)$ in expectation.

Notation. We want to construct hash functions $h : [u] \rightarrow [m]$. We use simple tabulation with an alphabet of $\Sigma$ 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 the 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 $1 - r^n$ 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. We begin by showing that simple tabulation preserves this guarantee. Building on this, we will show that the load of any fixed bin obeys Chernoff bounds. In Appendix A.1, we show that this holds even for a bin chosen as a function of the query hash code, $h(q)$.

Lemma 3. Suppose we hash $n \leq m^{1-\varepsilon}$ keys into $m$ bins, for some constant $\varepsilon > 0$. For any constant $\gamma$, there exists a constant $d = d(\varepsilon, c, \gamma)$, such that: with probability $\geq 1 - m^{-\gamma}$, all bins get less than $d$ keys.
Proof. We will show that no bin contains more than \( d = \min\{(\frac{1+\gamma}{2})^c, 2^{(1+\gamma)/c}\} \) elements. (As noted in the introduction, the fact that the constant \( d \) depends exponentially on \( \min\{c, \frac{1}{\epsilon}\} \) is unavoidable, but it is not a major obstacle in many applications.)

We will show that among any \( d \) elements, one can find a peelable subset of size \( t \geq \max\{d^{1/c}, \log d\} \). 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 \( \binom{n}{t} \) \( t \) 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: \( n^t/m^{t-1} = m^{(1-c)t}/m^{t-1} = m^{1-ct} \). To obtain failure probability \( m^{-\gamma} \), set \( t = (1+\gamma)/\epsilon \).

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

\[ \square \]

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 ordering \( \prec \) 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 \( \max_{\alpha \prec x} \alpha \) is fixed. For every \( \alpha \in [c] \times \Sigma \), we define the group \( G_\alpha = \{x \in S \mid \alpha = \max_{\alpha \prec x} \alpha\} \), the set of keys for whom \( \alpha \) is the last position-character to be fixed.

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

Lemma 4. There is an ordering \( \prec \) such that the maximal group size is \( \max_{\alpha} |G_\alpha| \leq n^{1-1/c} \).

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

We have to prove is that, as long as \( S \neq \emptyset \), there exists \( \alpha \) with \( |G_\alpha| \leq |S|^{1-1/c} \). If some position \( i \) has \( |\pi(S, i)| \geq |S|^{1/c} \), there must be some character \( \alpha \) 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 \( \alpha \) 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 \( G_\alpha \) is \( d \)-bounded if no bin contains more than \( d \) keys from \( G_\alpha \).

Lemma 5. Assume the number of bins is \( m \geq n^{1-1/(2c)} \). For any constant \( \gamma \), there exists a constant \( d_\gamma \) 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 \( G_\alpha \) in any bin, by Lemma 3. The conclusion follows by union bound over the \( \leq n \) groups.

Let \( X_\alpha \) be the number of elements from \( G_\alpha \) landing in the bin \( B \). We are quite close to applying Chernoff bounds to the sequence \( X_\alpha \), which would imply the desired concentration around \( \mu = n/m \). Two technical problems remain: \( X_\alpha \)-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 $G_\alpha$ 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 [MR95] holds even if the distribution of each $X_i$ is a function of $X_1, \ldots, X_{i-1}$, as long as $\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}]$ is a constant. We claim that this is the case: regardless of the hash codes for $\beta < \alpha$, $\mathbb{E}[\hat{X}_\alpha] = |G_\alpha|/m$.

Observe that whether or not $G_\alpha$ is $d$-bounded is known immediately before $h(\alpha)$ is fixed in the order $\prec$. Indeed, $\alpha$ is the last position-character to be fixed for any key in $G_\alpha$, so the hash codes of all keys in $G_\alpha$ 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 $\beta < \alpha$, 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.

## 3 Minwise Independence

We will prove that:

$$\frac{1}{n} \cdot \left(1 - \frac{O(\log n)}{n^{1/c}}\right) \leq \Pr[h(q) < \min h(X)] \leq \frac{1}{n} \cdot \left(1 + \frac{O(\log^2 n)}{n^{1/c}}\right) \quad (4)$$

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 $\ell = \gamma \log n$ for a large enough constant $\gamma$. Of particular interest is the minimum bin $[0, \frac{\ell}{n})$. We choose $\gamma$ sufficiently large for the Chernoff bounds of Theorem 1 to guarantee that the minimum bin in non-empty w.h.p.: $\Pr[\min h(X) < \frac{\ell}{n}] \geq 1 - \frac{1}{n^{\gamma}}$.

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 (4) continues to hold when the hash codes have $(1 + \frac{1}{\ell}) \log n$ bits, even if ties are resolved adversarially. Let $\tilde{h}$ be a truncation to $(1 + \frac{1}{\ell}) \log n$ 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:

$$\Pr[\tilde{h}(q) \leq \frac{\ell}{n} \land (\exists) x \in S : \tilde{h}(x) = \tilde{h}(q)] \leq \frac{\ell}{n} \cdot \left(n \cdot \frac{1}{n^{1+1/\ell}}\right) \leq \frac{O(\log n)}{n^{1+1/\ell}}$$

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

Both the lower and upper bounds start by expressing: $\Pr[h(q) < \min h(S)] = \int_0^{\ell/n} f(p)dp$, 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:

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

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 14. Let $h(\prec \alpha)$ done the choice for all position-characters $\beta \prec \alpha$. 

5
Remember that \( < \) 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 \( \alpha \), the hash 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 \( h(\prec \alpha) \), \( \Pr[h(z) \prec p \mid h(\prec \alpha)] = p \). By the union bound, \( \Pr[p \prec \min h(G_\alpha) \mid h(\prec \alpha)] \geq 1 - p \cdot |G_\alpha| \). This implies that:

\[
f(p) = \Pr[p \prec \min h(S) \mid h(q) = p] \geq \prod_{\alpha \succ q_c} (1 - p \cdot |G_\alpha|).
\] (5)

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

**Lemma 6.** Let \( p \in [0, 1] \) and \( k \geq 0 \), where \( p \cdot k \leq \sqrt{2} - 1 \). Then \( 1 - p \cdot k \geq (1 - p)^{(1+pk)^k} \).

**Proof.** First we note a simple proof for the weaker statement \((1 - pk) \leq (1 - p)^{(1+pk)^k}\). 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:

\[
(1 - p)^t \leq 1 - t \cdot p + \left( \frac{t}{2} \right)^2 p^2 < 1 - \frac{1}{(1 - \frac{p}{2})} \cdot t \cdot p
\]

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

We have just shown \( 1 - p \cdot k \geq (1 - p)^{(1+pk)^k} \). Removing the ceiling requires an “inclusion-exclusion” inequality with a non-integral number of experiments \( t \). Such an inequality was shown by Gerber [Ger68]: \((1 - p)^t \leq 1 - at + (at)^2 / 2\), even for fractional \( t \). Setting \( t = (1 + pk)k \), our result is a corollary of Gerber’s inequality:

\[
(1 - p)^t \leq 1 - pt + \frac{(pt)^2}{2} = 1 - p(1 + pk)k + \frac{1}{2} (p(1 + pk)k)^2 = 1 - pk - \frac{1 - \frac{p}{2}}{2} (p(1 + pk)k)^2 \leq 1 - pk.
\]

The lemma applies in our setting, since \( p < \frac{\ell}{n} = O\left(\frac{\lg n}{n}\right) \) and all groups are bounded \( |G_\alpha| \leq 2 \cdot n^{1-1/c} \). Note that \( p \cdot |G_\alpha| \leq \frac{\ell}{n} \cdot 2n^{1-1/c} = O\left(\ell n^{1/c}\right) \). Plugging into (5):

\[
f(p) \geq \prod_{\alpha \succ q_c} (1 - p \cdot |G_\alpha|) \geq \prod_{\alpha \succ q_c} (1 - p)^{|G_\alpha|(1+\ell/n^{1/c})} \geq (1 - p)^n(1+\ell/n^{1/c})
\]

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

\[
\Pr[h(q) < \min h(S)] = \frac{1}{m + 1} \int_0^{\ell/n} f(p) dp = 1 \frac{1 - (1 - \ell/n)^{m+1}}{m + 1} = \frac{n}{n + \frac{\ell}{\lg n}} - \frac{1}{n} \cdot \frac{O(\ell n^{1/c})}{n^{1/c}}
\]

**3.2 Upper bound**

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

\[
\Pr[h(q) < h(S)] \leq \Pr[\min h(S) \geq \frac{\ell}{n}] + \Pr[h(q) < h(S) \land h(q) < \frac{\ell}{n}] \leq \frac{1}{n^2} + \int_0^{\ell/n} f(p) dp
\]

To bound \( f(p) \), we will fix position-characters in the order \( < \) from Lemma 14, 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 $G_\alpha$ arranged themselves, when shifted randomly by $h(\alpha)$, they failed to land below $p$ with probability $1 - p |G_\alpha| \geq (1 - p)^{(1 + o(1))} |G_\alpha|$. 

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

To refine the analysis, we can use Lemma 3, which says that for $d = O(1)$, all groups $G_\alpha$ are $d$-bounded with probability $\geq 1 - \frac{1}{n^d}$. If $G_\alpha$ is $d$-bounded, its keys cannot cluster in less than $\lceil |G_\alpha|/d \rceil$ different bins.

When a group $G_\alpha$ 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 $R_\alpha$ be the set of representatives of $G_\alpha$. 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 $G_\alpha$ are decided up to a shift by $h(\alpha)$, and this common shift cannot change how keys cluster into bins. We obtain:

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

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, $|R_\alpha| \geq |G_\alpha|/d$, so we get $\Pr[p < \min h(G_\alpha) \mid h(\prec \alpha)] \leq (1 - p)^{|G_\alpha|/d}$ for almost all $h(\prec \alpha)$. Unfortunately, this is a far cry from the desired exponent, $|G_\alpha| \cdot (1 - O(n^{-1}/c))$.

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 $R_\beta$ just before character $\alpha$ is revealed: from any subset of $G_\beta$ 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 $\beta$ doesn’t change the clustering of $G_\beta$).

Let $C(\alpha)$ be the number of key pairs $(x, y)$ from the same group $G_\beta$ $(\beta \succ \alpha)$ 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 $\alpha^+$ be the successor of $\alpha$ 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 $n(\succ \alpha) = \sum_{\beta > \alpha} |G_\alpha|$ and define $n(\preceq \alpha)$ similarly. Our main inductive claim is:

**Lemma 7.** For any setting $(h(\prec \alpha))$ such that $h(q) = p$ and $\sum_{\beta \preceq \alpha} |R_\beta(\alpha)| = r$, we have:

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

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

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 Appendix A.2.) The lemma looks at the worst-case probability, over prior choices $h(\prec \alpha)$, that $p = h(q)$ remains the minimum among groups $G_\alpha, G_{\alpha+1}, \ldots$. After seeing the prior hash codes, the number of representatives in these groups is $r = \sum_{\beta \geq \alpha} |R_\beta(\alpha)|$. In the ideal case when $h(\alpha), h(\alpha+1), \ldots$ 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 \((1 - p)^{n(\alpha)}/(2d)\) 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 \(\hat{r} = \sum_{\beta > \alpha} R_\beta(\alpha)\). 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 \(O(\frac{n}{n} C(\alpha)/\hat{r})\). The weight of the worst case follows by \(\hat{r} \geq n(\alpha) / d\) and letting these terms accrue in the induction for \(\beta > \alpha\).

**Deriving the upper bound.** We now prove the upper bound on \(\Pr[h(q) < h(S)]\) assuming Lemma 7. Let \(\alpha_0\) be the first position-character fixed after the query. Since fixing the query cannot eliminate representatives,

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

**Lemma 8.** \(P(\alpha_0, p, n) \leq (1 - p)^n + (1 - p)^{n/(2d)} \cdot O(\lg^2 n)\).

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

Consider all pairs \((x, y)\) from the same group \(G_\gamma\), and order them by \(\beta = \max_\gamma (x \Delta y)\). 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 \(1/n(\beta)\), so this contribution is maximized if \(\beta\) immediately precedes \(\gamma\) in the order \(<\). That is, the sum is maximized when \(C(\beta) = \left\lceil \frac{G_{\beta + 1}}{2} \right\rceil\). We obtain \(A \leq \sum_{\beta} \frac{|G_{\beta + 1}|}{n(\beta)} \leq n^{1 - 1/c} \cdot \sum_{\beta} |G_{\beta}| / n(\beta)\). In this sum, each key \(x \in G_{\beta}\) contributes \(1/n(\beta)\), 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:

\[
\Pr[h(q) < h(S)] \leq \frac{1}{n^2} + \int_0^{\ell/n} \Pr[p < \min h(S) \mid h(q) = p] dp
\]

\[
\leq \frac{1}{n^2} + \Pr[(\exists \alpha) G_\alpha \text{ not d-bounded}] + \int_0^{\ell/n} P(\alpha_0, p, n) dp
\]

By Lemma 3, all groups are \(d\)-bounded with probability \(1 - \frac{1}{n^2}\). We also have \(\int_0^{\ell/n} (1 - p)^{n} dp = -\frac{(1 - p)^{n+1}}{n+1} \bigg|_{p=0}^{\ell/n} \leq \frac{1}{n+1}\). Thus, \(\Pr[h(q) < h(S)] \leq \frac{O(1)}{n^2} + \frac{1}{n+1} + \frac{1}{n/(2d)+1} \cdot O(\frac{\lg^2 n}{n^{1/c}}) = \frac{1}{n} \cdot \left(1 + \frac{O(\frac{\lg^2 n}{n^{1/c}})}{n^{1/c}}\right)\).

### 4 Analysis of Cuckoo Hashing

The proof is an encoding argument. A tabulation hash function from \(\Sigma^n \rightarrow [m]\) 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 \((2|\Sigma|^c \lg m) - \gamma\) 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 \(h_1\).
**An easy way out.** A subkey is a set of position-characters on distinct positions. If $a$ is a setkey, 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 such easy way out.

**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).

Let $v_0$ be a node of degree 3 in such an obstruction, and let its incident edges be $a$, $b$, $c$. The obstruction can be traversed by a walk that leaves $v_0$ on edge $a$, returns to $v_0$ on edge $b$, leaves again on $c$, and eventually meets itself. Other than visiting $v_0$ 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 $x_1 = a$. Technically, when the walk meets itself at the end, it is convenient to expand it with an extra key which is the one it used first 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 $x_{<i} = \bigcup_{j<i} x_j$ 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 $h_i = h_i \mod 2$. 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 is redundant and will be omitted. Indeed, we can compute this last hash code from the equality $h_i(x_i) = h_i(x_{i+1})$. This subroutine uses $(|\hat{x}_i \cup \hat{x}_{i+1}| - 1) \lg m$ 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, a reader will often need some context, e.g., to decode **COLL($x_i, x_{i+1}$),** one typically needs to know $i$, and the identities of $x_i$ and $x_{i+1}$ in $S$.

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

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

We observe that for each $i > 1$, we save $\varepsilon$ bits of entropy. Indeed, ID($x_i$) uses $\lg n$ bits, but COLL($x_{i-1}, x_i$) then saves $\lg m = \lg((1 + \varepsilon)n) \geq \varepsilon + \lg n$ 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 $\frac{1}{3} \lg n - O(1)$ bits, giving an overall saving of $\frac{1}{3} \lg n - O(1)$. Since we will use several such modifications, it is crucial to verify that they only touch distinct operations in the stream.

Each algorithmic modification to the stream of operations 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 already saved $\varepsilon k$ bits by the basic encoding ($\varepsilon$ 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 $\lg n - O(1)$.

**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 $x_{k+1} \subset x_{<k}$, 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 9** (safe-strong). If $\hat{x}_k \not\subset x_{k+1}$, we can save $\lg n - O(c \lg k)$ bits by changing $\text{HASHES}(x_k)$.

**Proof.** We can encode $\text{ID}(x_{k+1})$ using $c \lg k$ extra bits, since it consists only of known characters from $x_{<k}$. 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 provide us any cheap 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 10** (safe-weak). Assume that for any character sets $a, b$ on the same positions with $C(a), C(b) \geq n^{2/3}$, we have $h_0(a) \neq h_0(b)$ and $h_1(a) \neq h_1(b)$. If $\hat{x}_k \subset x_{k+1}$, we can save $\frac{1}{3} \lg n - O(c \lg k)$ bits by changing $\text{ID}(x_k)$.

**Proof.** The collision $h_k(x_k) = h_k(x_{k+1})$ means that $h_k(x_k \setminus \hat{x}_k) = h_k(x_{k+1} \setminus \hat{x}_k)$. But no frequent character sets collide, so either $|C(x_k \setminus \hat{x}_k)| \leq n^{2/3}$ or $|C(x_{k+1} \setminus \hat{x}_{k+1})| \leq n^{2/3}$. We encode which one is the case (one bit); assume the former by symmetry. To encode $\text{ID}(x_k)$, we first encode $x_k \setminus \hat{x}_k$ using $c \lg k$ bits, since it consists only of known characters. Then, we can encode $\hat{x}_k$ by indicating a choice out of $C(x_k \setminus \hat{x}_k)$. This only requires $\frac{2}{3} \lg n$ bits, which gives the desired saving.

**Piggybacking.** While the ideas from above 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 11** (odd-size saving). Consider two edges $e, f$ and an $i$ satisfying:

$$h_{i+1}(e) = h_{i+1}(f); \quad e \setminus x_{\leq i} \neq f \setminus x_{\leq i}; \quad e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1}.$$  
We can save $\lg n - O(c \lg k)$ bits by changing $\text{COLL}(x_{i+1}, x_{i+2})$.

**Lemma 12** (piggybacking). Consider two edges $e, f$ and an $i$ satisfying:

$$h_i(e) = h_i(f); \quad e \setminus x_{\leq i} \neq f \setminus x_{\leq i}; \quad e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1}.$$  
We can encode $\text{ID}(e)$ and $\text{ID}(f)$ using only $O(c \lg k)$ bits, after modifications to $\text{ID}(x_i), \text{ID}(x_{i+1})$, and $\text{COLL}(x_i, x_{i+1})$.

The difference between the lemmas is the parity (side in the bipartite 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 $\text{ID}(e)$ and $\text{ID}(f)$ almost for free: we say $e$ and $f$ piggyback on the encodings of $x_i$ and $x_{i+1}$.

Before we prove the lemmas, we demonstrate their use 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 11 saves a $\lg n$ term. (Note that $i \leq k - 2$, so the lemma modifies a valid operation of the stream.)

If $i$ has the same parity as $k$, Lemma 12 gives us the identity of $x_k$ at negligible cost. Then, we can remove the operation $\text{ID}(x_k)$ from the stream, and save $\lg n$ bits. (Again, note that $i \leq k - 2$, so the ID we remove is different from the ones we piggyback on.)
Proof of Lemma 11. Since \( e \) and \( f \) have different free characters before \( \hat{x}_{i+1} \), but identical free characters afterward, it must be that \( (e \cap \hat{x}_{i+1}) \Delta (f \cap \hat{x}_{i+1}) \) is nonempty. The last character of this set, in any order, is redundant (i.e. it can be omitted, saving \( \log m \) bits). This is because \( h_{i+1}(e \cap x_{\leq i+1}) = h_{i+1}(f \cap x_{\leq i+1}) \), which follows from \( e \) and \( f \) having identical free characters after \( x_{i+1} \). The last character of \( \hat{x}_{i+2} \) is also redundant due to the collision of \( x_i \) and \( x_{i+1} \). Thus, we can save \( 2 \log m \) bits instead of the usual \( \log m \) bits.

Proof of Lemma 12. We begin with two simple cases that allow us to save a character hash code, as in the analysis above. If the following claim gives us a saving of \( \log n \), we can afford to encode ID(\( e \)). Then ID(\( f \)) can be encoded using \( O(c \log k) \) bits, since \( f \) differs from \( e \) only in position-characters from \( x_{\leq i+1} \).

Claim 13. We can save \( \log n - O(c \log k) \) bits by changing \( \text{COLL}(x_{i-1}, x_i) \) if one of the following happens:
1. \( x_{i+1} \) contains new characters outside \( e \) and \( f \): \( \hat{x}_{i+1} \setminus (e \cup f) \neq \emptyset \).
2. the new characters of \( x_i \) outside \( e \cup f \) are not entirely repeated in \( x_{i+1} \): \( \hat{x}_i \setminus (e \cup f) \setminus x_{i+1} \neq \emptyset \).

Proof. As above, we remark that \((e \cap \hat{x}_{i+1}) \Delta (f \cap \hat{x}_{i+1})\) is nonempty. Furthermore, its last character is redundant, since \( h_i((e \cap x_{\leq i+1}) = h_i((f \cap x_{\leq i+1}) \). If we can find an order on the characters \( \hat{x}_i \cup \hat{x}_{i+1} \) in which another character is redundant, we will save \( 2 \log m \) bits, compared to the regular COLL encoding that save \( \log m \).

Another redundancy exists if \( \hat{x}_{i+1} \) contains a character \( \alpha \) outside \( e \cup f \), or if \( \hat{x}_i \) contains a character \( \alpha \) outside \( e \cup f \) that is not repeated in \( x_{i+1} \). In both of these cases, the redundancy comes from collision of \( x_i \) and \( x_{i+1} \), if we place \( \alpha \) last in the order.

If the claim does not apply, we have a lot of structure on the keys. In particular, we can write ID(\( x_i \)), ID(\( x_{i+1} \)), ID(\( e \)), and ID(\( f \)) using \( 2 \log n + O(c \log k) \) bits in total. Let \( \xi = \hat{x}_i \setminus (e \cup f) \) and \( \hat{e} = e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1} \). We write:
- the coordinates on which \( \xi \) and \( \hat{e} \) appear, taking \( 2c \) bits.
- the value of \( \xi \) using Huffman coding. We consider the projection of all \( n \) keys on the coordinates of \( \xi \). In this distribution, \( \xi \) has frequency \( \frac{|\xi|}{n} \), so its Huffman code will use \( \log \frac{n}{|\xi|} + O(1) \) bits.
- the value of \( \hat{e} \) using Huffman coding. This uses \( \log \left( \frac{n}{|C(\hat{e})|} \right) + O(1) \) bits.
- if \( |C(\xi)| \leq |C(\hat{e})| \), we write \( x_i \) and \( x_{i+1} \). Each of these requires \( \lceil \log_2 |C(\xi)| \rceil \) bits, since \( \xi \subset x_i, x_{i+1} \) and there are \( |C(\xi)| \) completions of \( \xi \) to a full key. Using an additional \( O(c \log k) \) bits, we can write \( e \cap x_{\leq i+1} \) and \( f \cap x_{\leq i+1} \). Remember that we already encoded \( \hat{e} = e \setminus x_{\leq i+1} = f \setminus x_{\leq i+1} \), so the decoder can recover \( e \) and \( f \).
- if \( |C(\hat{e})| < |C(\xi)| \), we write \( e \) and \( f \), each requiring \( \lceil \log_2 |C(\hat{e})| \rceil \) bits. Since we know \( \xi = \hat{x}_i \setminus (e \cup f) \), we can write \( x_i \) using \( O(c \log k) \) bits: write the old characters outside \( \hat{x}_i \), and which characters of \( e \cup f \) to use in \( \hat{x}_i \). If Claim 13 fails, \( \hat{x}_{i+1} \subset e \cup f \), so we can also write \( x_{i+1} \) using \( O(c \log k) \).

Overall, the encoding uses space:
\[
\log \frac{n}{|C(\xi)|} + \log \frac{n}{|C(\hat{e}_{i+1})|} + 2 \log \min \{ |C(\xi)|, |C(\hat{e}_{i+1})| \} + O(c \log k) \leq 2 \log n + O(c \log k)
\]

Finale. The rest of the proof shows how to obtain a saving of at least \( \frac{1}{2} \log n - O(c \log k) \) by a careful combination of the above techniques. It appears in Appendix A.3

References


A Omitted Proofs

A.1 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 $\prec$:

\[\text{A Omitted Proofs}
\text{A.1 The Load of a Query-Dependent Bin}
\]

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 14. 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 4. Each time we select the position-character \( \alpha \) minimizing \( |G_\alpha| \), place \( \alpha \) next in the order \( \prec \), and remove \( G_\alpha \) 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 \( \alpha \) 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 \( \alpha \) 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 \( \alpha \) on position \( i \), we have \( |G_\alpha| \leq \prod |\pi(S, j)| \leq |S|^{1-1/c} \).

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 \( X_\alpha \) be the contribution of \( G_\alpha \) to bin \( B_q \), we see that the distribution of \( X_\alpha \) is determined by the hash codes fixed previously (including the hash code of \( q \), fixing the choice of the bin \( B_q \)). But \( \mathbb{E}[X_\alpha] = |G_\alpha|/m \) holds irrespective of the previous choices. Thus, Chernoff bounds continue to apply to the size of \( B_q \).

A.2 Proof of Lemma 7

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

\[
A = \Pr \left[ p < \min \bigcup_{\beta \geq \alpha} h(G_\beta) \land (\forall \alpha)G_\alpha \text{ } d\text{-bounded } | h(\prec \alpha) \right]
\]

If for some \( \beta \), \( |R_\beta(\alpha)| < |G_\alpha|/d \), 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:

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

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

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

\[
A_2 \leq \sum_{\delta = 0}^{\Delta_{\text{max}}} \Pr[\Delta = \delta \mid h(\prec \alpha), p > \min h(G_\alpha)] \cdot P(\alpha^+, p, \hat{r} - \delta)
\]
where we had \( P(\alpha^+, p, \hat{r} - \delta) = (1 - p)^{\hat{r} - \delta} + (1 - p)^{n(\beta)}/(2d) \cdot \sum_{\beta > \alpha} 4C(\beta) \cdot (\ell/n) \cdot n(\beta)/\Delta. \)

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

\[
A_2 \leq A_3 + (1 - p)^{n(\beta)}/(2d) \cdot \sum_{\beta > \alpha} 4C(\beta) \cdot (\ell/n) \cdot n(\beta)/\Delta
\]

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

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

**Observation 15.** We have: \( \mathbb{E}[\Delta | h(\beta), p > \min h(G_\alpha)] \leq 2 \cdot C(\alpha) \cdot \frac{\ell}{n}. \)

**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. \( \alpha = \max (x \Delta y) \). If \( h(\alpha) \) is uniform, such a pair \((x, y)\) collides with probability \( \frac{\ell}{n} \), regardless of \( h(\beta) \). By linearity of expectation \( \mathbb{E}[\Delta | h(\beta)] \leq C(\alpha) \cdot \frac{\ell}{n}. \)

However, we have to condition on the event \( p > \min h(G_\alpha) \), which makes \( h(\beta) \) 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 \( \Pr[\Delta = \Delta_{\text{max}}] = (2 \cdot C(\alpha) \cdot \frac{\ell}{n})/\Delta_{\text{max}} \), and all the rest of the mass is on \( \Delta = 0 \). This gives:

\[
A_3 \leq (1 - p)^{\hat{r}} + \frac{2 \cdot C(\alpha) \cdot (\ell/n)}{\Delta_{\text{max}}} \cdot (1 - p)^{\hat{r} - \Delta_{\text{max}}}
\]

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

\[
A_3 \leq (1 - p)^{\hat{r}} + \frac{2 \cdot C(\alpha) \cdot (\ell/n)}{\Delta_{\text{max}}} \cdot (1 - p)^{n(\beta)/(2d)}
\]

\[
\implies A_2 \leq (1 - p)^{\hat{r}} + (1 - p)^{n(\beta)/(2d)} \cdot \sum_{\beta > \alpha} 4C(\beta) \cdot (\ell/n) \cdot n(\beta)/d
\]

\[
\implies A \leq (1 - p)^{|R_\alpha|} \cdot (1 - p)^{\hat{r} - |R_\alpha|} + (1 - p)^{n(\beta)/(2d)} \cdot \sum_{\beta > \alpha} 4C(\beta) \cdot (\ell/n) \cdot n(\beta)/d
\]

This completes the proof of Lemma 7, and the bound on minwise independence.

**A.3 Cuckoo Hashing: Assembling the Pieces**

We now show how to obtain a saving of at least \( \frac{4}{3} \log n - O(c \log k) \) by a careful combination of the above techniques.

We first consider the case when \( b \setminus x_{\leq k} \neq \emptyset \) or \( c \setminus x_{\leq k} \neq \emptyset \). Let \( y_1 = b \) if \( b \) contains free characters, and \( y_1 = c \) if \( c \) contains free characters. We consider a walk \( y_1, y_2, \ldots \) along the original walk of the obstruction. (If starting at \( b \), we follow the original walk in reverse.) Let \( \bar{y}_i = y_i \setminus x_{\leq k} \setminus y_{\leq i} \) 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 \( \hat{y}_{t+1} = \emptyset \). Our encoding of the second walk will be:

\[
\text{ID}(y_1); \text{COLL}(x_1, y_1); \text{ID}(y_2); \text{COLL}(y_2, y_1); \ldots; \text{ID}(y_t); \text{COLL}(y_t, y_{t-1}) \text{HASHES}(h_t, y_t)
\]

We can use safe savings to obtain a saving of \( \frac{1}{3} \lg n \) bits from this second walk. Remember that Lemmas 9 and 10 only modify \( \text{HASHES}(h_t, y_t) \) or \( \text{ID}(y_t) \). These do not interact with the first walk, so we can use any technique (including piggybacking) to saving \( \lg n \) bits there. We obtain a total saving of \( \frac{1}{3} \lg n - O(1) \).

We are left with the situation \( b \setminus x_{\leq k} = c \setminus x_{\leq k} = \emptyset \), which requires a detailed case analysis.

**Lemma 16.** Let \( t \) be such that \( b \setminus x_{< t} \neq \emptyset \) but \( b \setminus x_{\leq t} = \emptyset \). We can save \( \lg n - O(c \lg k) \) bits by modifying \( \text{COLL}(x_t, x_{t-1}) \) if \( t \) is odd, or \( \text{COLL}(x_t, x_{t+1}) \) if \( t \) is even. The same holds if \( b \) is replaced by \( c \).

**Proof.** The idea is to save an additional character hash code (\( \lg m \) bits) when writing \( h_0(x_i) \), since we know \( h_0(a) = h_0(b) = h_0(c) \). We modify \( \text{COLL}(x_t, x_{t+1}) \), where the \( \pm 1 \) choice depends on the parity of \( t \) as in the lemma statement. First we write the identity of \( b \), which takes \( O(c \lg k) \) bits. Then we write the characters of \( \hat{x}_t \). The last character from \( \hat{x}_t \cap b \) is redundant, since the decoder knows \( h_0(b) \). Finally we write the characters of \( \hat{x}_{t+1} \), the last of which is redundant due to collision with \( x_t \).

Let \( t \) be the first time when \( b \setminus x_{< t} = \emptyset \) or \( c \setminus x_{< t} = \emptyset \) (or both). If \( t \leq k - 1 \), Lemma 16 can be applied in conjunction with any safe saving, since the safe saving only requires modifications to \( \text{ID}(x_k) \) or \( \text{HASHES}(h_k, x_k) \). When \( t = k \), this combination only works if \( k \) is odd, since then Lemma 16 chooses to modify \( \text{COLL}(x_{k-1}, x_k) \).

The remaining case is when \( k \) is even and the last characters of \( b \) and \( c \) are revealed by \( \hat{x}_k \). \( b \cap \hat{x}_k \neq \emptyset \) and \( c \cap \hat{x}_k \neq \emptyset \).

**Lemma 17.** If \( b \cap \hat{x}_k \neq c \cap \hat{x}_k \), we can save \( 2 \lg n - O(c \lg k) \) bits by modifying \( \text{HASHES}(h_k, x_k) \).

**Proof.** Assume by symmetry that \( |b \cap \hat{x}_k| \geq |c \cap \hat{x}_k| \). Since the sets are different, there exists a character \( \alpha \in b \cap \hat{x}_k \) which is not in \( c \cap \hat{x}_k \). We reveal the \( h_k \) hash codes in any order that leaves \( \alpha \) last. The last character of \( c \) is redundant, since we know \( h_0(c) \). Also, \( \alpha \) is redundant since we know \( h_0(b) \).

It remains to deal with \( b \cap \hat{x}_k = c \cap \hat{x}_k \). We can save \( \lg n \) bits by modifying \( \text{HASHES}(h_k, x_k) \), since the last character of the set \( b \cap \hat{x}_k = c \cap \hat{x}_k \) is redundant. If \( \hat{x}_k \setminus b \setminus x_{k+1} \neq \emptyset \), this gives us a second redundant character in \( \text{HASHES}(h_k, x_k) \), so we are done.

We will now apply the piggybacking machinery to \( b \) and \( c \). Note that \( h_0(b) = h_0(c) \), and \( b \setminus x_{< k} = c \setminus x_{< k} \). Thus, there exists \( i < k - 1 \) such that \( b \setminus x_{\leq i} \neq c \setminus x_{\leq i} \) but \( b \setminus x_{\leq i+1} = c \setminus x_{\leq i+1} \). If \( i \) is odd, Lemma 11 (odd-side saving) can save \( \lg n \) bits by modifying \( \text{COLL}(x_{i+1}, x_{i+2}) \). Given that \( k \) is even, \( i < k - 1 \), and \( i \) is odd, we must have \( i < k - 3 \), so these do not go out of bounds.

If \( i \) is even, Lemma 12 (piggybacking) can give use \( \text{ID}(a) \) and \( \text{ID}(b) \) at a cost of just \( O(c \lg k) \) bits. The last operation in the sequence that the Lemma can modify is \( \text{ID}(x_{i+1}) \). Since \( i + 1 < k \), this does not conflict with \( \text{ID}(k) \).

If \( \hat{x}_k \setminus b = \emptyset \), we can encode \( \text{ID}(x_k) \) with just \( O(c \lg k) \) bits once \( b \) is known, so we can save \( \lg n - O(c \lg k) \) bits in \( \text{ID}(x_k) \).

This leaves us with the case \( \hat{x}_k \setminus b \neq \emptyset \) and \( \hat{x}_k \setminus b \subset x_{k+1} \). Since we have revealed \( b \) (by piggybacking), we now satisfy the conditions of Lemma 10 (safe-weak saving). This is the final case of our analysis, and the most comprehensive: we use both piggybacking and safe-weak (no easy way out) to compress an ID.

### B Experimental evaluation of hash functions.

In this section, we make some simple experiments comparing simple tabulation with other hashing schemes. Our experiments are essentially the same as those in [TZ09] except that we here include simple tabulation whose relevance was not realized in [TZ09].
Recall that our point was to analyze simple tabulation which is known as an easy to implement practical hashing scheme. Even though simple tabulation is not 4-independent, we proved that it in several algorithmic applications shared some of the qualities known only for hash functions with much higher independence. This contrasts previous negative findings for other practical schemes.

As our main application, we consider linear probing which in general is known to require 5-independence for expected constant operation time. Of other practical hashing schemes, we consider the previously mentioned multiplication-shift schemes: both the universal (Univ-mult-shift) [DHKP97] and 2-independent (2-indep-mult-shift) [Die96] versions. We also compare with previous 5-independent hashing schemes, both the classic based on Mersenne primes (5-indep-Mersenne-prime), and the 5-independent tabulation scheme of Thorup and Zhang (5-indep-TZ-table) [TZ04, TZ09].

Below, in Table 1 we present timings for the different schemes, first mapping 32-bit keys to 32-bit values, second mapping 64-bit keys to 64-bit values. For the 2-independent multiplication-shift scheme on 64 bits, we employ a cross product trick to get down two 64-bit multiplications (see [Tho09] for details). For the Mersenne prime scheme, we use the prime $2^{61} - 1$ for 32 bits and $2^{81} - 1$ for 64 bits. As mentioned, the essential difference between our experiments and those in [TZ09] is that simple tabulation is included. Computer A uses a 32-bit processor while computer B uses a 64-bit processor. Not surprisingly, we see that Computer B relatively speaking is relatively faster when 64-bit multiplications are critical (univ-mult-shift for 64 bits, 2-indep-mult-shift, and 5-indep-Mersenne-prime).

Our interest here is the performance of simple tabulation. In the case of 32-bits, we see that it in both computers has performance 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 get 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, and then we may no longer fit in fastest cache. On 64-bit, we see that 2-indep-mult-shift is twice as fast, but this is partly due to the special fast implementation mentioned above.

As discussed previously, we can normally hash longer strings down to 64 bits. The essential point we get is that simple tabulation is at least a factor 2 faster than the higher independence schemes.

We now consider what happens when we plug the schemes into the context of linear probing where 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 over 10 million insert/delete

<table>
<thead>
<tr>
<th>bits</th>
<th>algorithm</th>
<th>hashing time (nanoseconds)</th>
<th>computer A</th>
<th>computer B</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>univ-mult-shift</td>
<td>1.87</td>
<td>2.33</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2-indep-mult-shift</td>
<td>5.78</td>
<td>2.88</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>5-indep-Mersenne-prime</td>
<td>99.70</td>
<td>45.06</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>5-indep-TZ-table</td>
<td>10.12</td>
<td>12.66</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>simple-table</td>
<td>4.98</td>
<td>4.61</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>univ-mult-shift</td>
<td>7.05</td>
<td>3.14</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>2-indep-mult-shift</td>
<td>22.91</td>
<td>5.90</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>5-indep-Mersenne-prime</td>
<td>241.99</td>
<td>68.67</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>5-indep-TZ-table</td>
<td>75.81</td>
<td>59.84</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>simple-table</td>
<td>15.54</td>
<td>11.40</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Average time per hash computation for 10 million hash computations on Computer A (single-core Intel Xeon 3.2 GHz 32-bit processor with 2048KB cache), and B (dual-core Intel Xeon 2.6 GHz 64-bit processor with 4096KB cache).
cycles. We ran 100 such experiments, with different random seeds for the hash function, and the plots show the distribution of these averages.

First we consider the case of random input, hence where the randomization properties of the hash function are irrelevant. The interesting thing then is how the different schemes behave when they have to compete with the linear probing table for the cache. All schemes have a common high additive cost for a random access to the hash location in the linear probing table, and this decreases the relative difference in performance. These results are presented in Figure 2. Next we consider the case where the input keys are structured in the

Figure 2: Random keys. The quality of the hash function does not matter so the schemes have very similar distribution of number of probes, but update times different due to different times to compute hash functions on Computer A.

sense of being drawn from a dense interval: a commonly occurring case in practice which is known to cause unreliable performance for most simple hashing schemes [PPR09, PT10].

C Linear probing

We use simple tabulation for a linear probing table with $t$ slots. Here as for other bins, we assume that $t$ is a power of two. Let $\alpha = (1 - \varepsilon) = n/t$ be the fill.

The displacement from $q$ is limited by the size (in slots) of the maximal filled interval $I$ that includes $h(q)$. Here full means that $#\{x \in S | x \in I\} \geq |I|$. Fundamentally we will argue that w.h.p., our perfor-
Figure 3: Keys from dense interval. The multiplication-shift schemes sometimes use far more probes, which also shows in much longer times on Computer A and B.
mance is within constant factors of that with a truly random function.

**When \( \varepsilon \) medium** We will now consider the case where the fill \( \alpha = (1 - \varepsilon) \) is bounded from both 0 and 1. For this case we have a very simple argument. We want to bound the probability of a filed interval \( I \) of length at least \( x \) containing \( q \). When an interval is full it has more than \((1 + \varepsilon)\) times as many keys as expected. The issue is that \( I \) could be placed many ways. Also note that there may be no small filled interval containing \( q \), but only longer ones.

Let \( x \) be rounded so that \( \varepsilon x \) is a power of two. We only consider cases where \( x > 1/\varepsilon \). It is easy to see that if there is a filed interval \( I \) of length at least \( x \) containing \( q \), then for some integer \( i \geq 0 \), if we consider bins of size \( 2^i \varepsilon x/t \), then there has to be at least one of the \( 4/\varepsilon \) bins around \( q \) which has is \((1 + \varepsilon/2)\) times its expected mean. By (2), we get an overall probability bound of

\[
\sum_{i \geq 0} 1/ \exp(\Omega(2^i x \varepsilon n/t)) = 1/ \exp(\Omega(x)).
\]

Below we will consider cases where \( \varepsilon = o(1) \) and \( \alpha = o(1) \).

However, when \( \varepsilon = o(1) \), we can do a lot better. In fact, for the expectation, we will match the \( O(1/\varepsilon^2) \) bound of Knuth for truly random hash functions, improving on the \( O(1/\varepsilon^3) \) bound of Pagh et al.

**When \( \varepsilon \) is small** We now consider the case where \( \varepsilon = o(1) \). We will get an expected run length of \( O(1/\varepsilon^2) \), matching within a constant factor the bound of Knuth for truly random hash functions, and improving on the \( O(1/\varepsilon^3) \) bound of Pagh et al. for general 5-independent hashing.

We say that interval \( I \) has dyadic level \( i \) if \( i \) is the largest index such that \( I \) contains a dyadic interval of size \( 2^i \), that is, for some offset \( o \), \([o2^i, (o+1)2^i] \subseteq I \). Then \( |I| < 2^{i+2} \). Moreover, we note that \( I \) is contained in the union \( U \) of the 3 dyadic intervals of size \( 2^{i+1} \) around \( h(q) \).

With the standard decomposition of \( I \) into maximal dyadic intervals, we get that \( I \) is composed of up to two dyadic intervals on levels \( j = 0, \ldots, i \).

We know that the total excess over the mean in \( I \) is at least \( \varepsilon |I| \geq \varepsilon 2^i \). We now present a way to divide this minimal excess on the dyadic decomposition. We give the up to two dyadic intervals on level \( j \) a target excess of \( \Delta_j = \varepsilon 2^i/2^{(i-j)/4} \). Then the total target excess is at most

\[
2 \sum_{j=0}^i \Delta_j = 2 \sum_{j=0}^i \varepsilon 2^i/2^{(i-j)/4} < \varepsilon 2^i 2(2 + 2^{3/4} + 2^{1/2} + 2^{1/4})/21 < \varepsilon 2^i 12.58/21 < \varepsilon 2^i.
\]

Since this is less than \( \varepsilon 2^i \), at least one of the dyadic intervals must exceed the target. In order to apply (2), it is convenient to only consider \( j \) such that \( \Delta_j \leq 2^i \). Let \( j_0 \) be the smallest such \( j \). Instead of looking for excess in the intervals below level \( j_0 \), we look for excess in the extra dyadic level \( j_0 \) interval on either side. Since this two extra intervals are not fully included in \( I \), we view their full contents as excess for \( I \), that is, \( \Delta_{j_0} + (1 - \varepsilon)2^i < 2^{j_0+1} \). Thus, the total target excess for \( I \) is \( 2^{j_0+2} + 2 \sum_{j=j_0+1}^i \Delta_j \). By definition, \( 2^{j_0-1} < \Delta_{j_0-1} < \varepsilon 2^i/21 \), to the total is

\[
2^{j_0+2} + 2 \sum_{j=j_0+1}^i \Delta_j < \varepsilon 2^i 12.58/21 + \varepsilon 2^i 8/21 = \varepsilon 2^i 20.58/21 < \varepsilon 2^i.
\]

Again this is less than \( \varepsilon 2^i \), so at least one of the relevant dyadic intervals must exceed the target.

For the application of (2), for dyadic level \( j \) intervals, we have \( \mu = (1 - \varepsilon)2^j \), \( \delta = \Theta(\varepsilon 2^{j(i-j)/2}) \), and \( \Theta = \Theta(2^j) \). Hence for a given level \( j \) dyadic interval \( J \), by (2), the probability of exceeding the target is bounded by

\[
\Pr[X > x] < 1/ \exp(\Omega((\varepsilon 2^{j(i-j)/2} 2^j)) = 1/ \exp(\Omega((\varepsilon 2^{j(i-j)/2} 2^j)).
\]

19
Next we note that to fill any interval around \( h(q) \) of dyadic level \( i \), there must be some \( j \in \{0, \ldots, i\} \) such that some dyadic level \( j \) interval \( J \) has an excess at least \( \Delta_j \). Moreover, \( J \) is contained in the union the 3 dyadic intervals of size \( 2^{i+1} \) around \( h(q) \). This means that there are at most \( 3 \times 2^{i-j} \) dyadic level \( j \) intervals that are used in the dyadic decomposition of any interval of dyadic level \( i \). Combining with the above formulas, we get that the probability of excess for any dyadic level \( j \) interval is
\[
3 \times 2^{i-j} / \exp(\Omega((\varepsilon^2 2^{i+\frac{1}{2}(i-j)}))
\]
Our total error probability is
\[
\sum_{j=j_0}^{i} 3 \times 2^{i-j} / \exp(\Omega((\varepsilon^2 2^{i+\frac{1}{2}(i-j)}))
\]
The last \( j = i \) term dominates, so the overall error probability is \( O(1/ \exp(\Omega((\varepsilon^2 2^i))) \) which bounds the probability of a full interval \( I \) of dyadic level \( i \) containing \( h(q) \). This falls rapidly when \( 2^i > 1/\varepsilon^2 \). We conclude that the probability of a displacement by \( x \) is
\[
\Pr[X > x] < 1/ \exp(\Omega(\varepsilon^2 x)).
\]

Small \( \alpha = (1 - \varepsilon) \) We just consider \( \alpha < \frac{1}{4} \). In this case, we again consider the maximal covered dyadic interval of size \( 2^i \). We know that within the whole interval \( I \), we have excess at least \( \varepsilon |I| > \frac{\alpha}{2} 2^i \). We also know that \( J \) is contained in three level \( i \) dyadic intervals, which should hence have at least \( \varepsilon |I| > \frac{\alpha}{2} 2^i \) keys. The intervals themselves have a total mean of \( \frac{3}{4} \) keys, so we have to account for an extra excess of \( \frac{\alpha}{4} 2^i \) keys, or \( \frac{\alpha}{2} 2^i \) keys for one of the three relevant dyadic level \( 2^i \) interval. This means that we can apply (3), hence that the probability of displacement by \( x \) is bounded by
\[
\Pr[X > x] < (\alpha / x) \Omega(x).
\]
The above bound is only good when \( x = \omega(1) \). For \( x = O(1) \), we get a better bound from 2-independence of \( O(\alpha) \).

D Hard Instance in Cuckoo Hashing

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 \( h_1(a) = h_1(b) = h_1(c) \);
- there exist \( x, y \in [n^{1/3}] \) with \( h_2(x) = h_2(y) \).

If both happen, then the elements \( ax, ay, bx, by, cx, cy \) cannot be hashed. Indeed, on the left side \( h_1(a) = h_1(b) = h_1(c) \) so they only occupy 2 positions. On the right side, \( h_2(x) = h_2(y) \) so they only occupy 3 positions. In total they occupy 5 < 6 positions.

The probability of 1. is roughly \( (n^{2/3})^3/n^2 = \Omega(1) \). This is because tabulation (on two characters) is 3-independent.

The probability of 2. is roughly \( (n^{1/3})^2/n = 1/n^{1/3} \).

So overall tabulation-based cuckoo hashing fails with probability \( \Omega(1/\sqrt{n}) \).

D.1 Bloomier Filters

We want a static dictionary containing \( n \) data items, each with \( k \) bits of associated information. If we only care about retrieving data for elements in the set (and never run membership queries), the optimal space is \( \Theta(nk) \) bits. This can be achieved by the power of two choices: with constant probability, the graph induced by the two hash functions have no cycle. So we can solve a system of equations and set the values in the two arrays, making the data associated to \( x \) be \( A[h_1(x)] \oplus B[h_2(x)] \).
The following lemma suggests that, for Bloomier filters with simple tabulation, one shouldn’t try to reach a state with zero cycles. Instead, we will later upper bound the expected number of cycles to \( f(c) \). Thus, you should just use a stash of constant size to break the cycles.

**Lemma 18.** The expected number of cycles is \( 2^{\Omega(c)} \).

**Proof.** Our set is the \( c \)-dimensional cube \([n^{1/c}]^c \). Let \( a, b \) be half-keys on some \( c/2 \) positions, at maximal Hamming distance \( c/2 \). Let \( c, d \) be half-keys on the other \( c/2 \) positions at maximal Hamming distance. We will look for length-4 cycles of the form \( ab \sim ac \sim db \sim dc \). These form a cycle iff \( h_1(a) = h_1(d) \) and \( h_2(b) = h_2(c) \), which happens with probability \( 1/(2n)^2 \). For a fixed set of character positions, there are \( \sqrt{n} \) choices for \( a \) and \( (n^{1/c} - 1)^{c/2} = (1 - o(1)) \cdot \sqrt{n} \) choices for \( b \). Similarly for \( (c, d) \). Thus, for a fixed partitioning of the characters into two sets, we expect \( 1/4 - o(1) \) cycles of this form. There are \( \binom{c/2}{c/2} \) partitioning. We know we are not double-counting any cycle, since \((a, b) \) and \((c, d) \) are at maximal Hamming distances. So we expect \( 2^{\Omega(c)} \) cycles of this special form.

**Theorem 19.** Bloomier filters with a stash of \( f(c) \) can be constructed with \( \Omega(1) \) probability.

**Proof.** We have shown that if tables have load \( 2^{-\alpha} \), the probability that there exists a chain of length \( k \) is \( \leq \max\{O\left(\frac{1}{n}\right), 2^{k\alpha - O(c \lg k)}\} \).

Using the normal load of \( \frac{1}{2} \), we deduce that, with probability \( 1 - \epsilon \), there is no bad chain of length \( \geq c \lg c \). Now it remains to deal with bad chains of length \( O(c \lg c) \).

We claim that, with probability \( 1 - \epsilon \), the maximum number of disjoint bad chains is at most some \( f(q) \). Then, the following algorithm can construct a Bloomier filter with stash \( f(q) \): find a chain; move it all to stash; repeat. If the stash fills, rehash.

Assume for contradiction that, with probability \( \epsilon \), there exist \( f(c) \) disjoint bad chains. Then, with probability \( \geq 1 - 2\epsilon \), there exist at least \( f(c) \) disjoint bad chains of length \( O(c \lg c) \). We subsample the set of keys randomly, keeping a fraction \( 2^{-c} \) of them. Fix the hash functions to some arbitrary values. After subsampling, each chain of length \( k \) survives with probability \( 2^{-ck} \), so any chain survives with probability \( 2^{-O(c^2 \lg c)} = c^{-O(c^2)} \). Since the chains are disjoint, so these probabilities are independent. Thus, if there are \( C \geq c^{O(c^2)} \) chains, one survives with probability \( 1 - \epsilon \). Averaging over the hash functions and the subsampling, a bad chain remains with probability at least \( 1 - 3\epsilon \). So we can fix the subsampling and maintain this probability. That is, there exists a fixed set of size \( n/2^c \) that has a bad chain with probability \( 1 - 3\epsilon \). But this contradicts the encoding bound, which shows that no cycles of any length \( k \geq 2 \) exists with probability more than \( \epsilon \) when the density is so small. \( \square \)

### E Fourth Moment Bounds

Assume \( n \) items \( \{x_1, \ldots, x_n\} \) are placed into \( b \) bins using a hash function \( h \). In the interest of general load balancing, we assume each item has a weight \( w_i \geq 0 \). The first concern in the analysis of hash functions is to bound the (weight of the) elements which land in a fixed bin (say, bin 0). Let \( X_i \) be the indicator of the event \( \{h(x_i) = 0\} \). The weight of our bin will then be \( W = \sum_i w_i \cdot X_i \).

For many applications, one needs a relatively strong concentration of \( W \) around \( \mu = \mathbb{E}[W] \), which in obtained by looking at the 4th moment of \( W \), defined as \( \mathbb{E}[(W - \mu)^4] \). The point in bounding the 4th moment is that \( \Pr[W > \mu + \Delta] \leq \mathbb{E}[(W - \mu)^4]^{\Delta^4} \).

When \( h \) is 4-independent, the correct bound on the 4th moment is given by:

\[
\mathbb{E}[(W - \mu)^4] = O\left(\frac{1}{b} \sum_i w_i^4 + \frac{1}{b^2} \left( \sum_i w_i^2 \right)^2 \right)^2
\]

Plugging in \( w_i = 1 \) for intuition, we see that the bound is \( O\left(\frac{n^2}{b^2}\right) = O(\mu^2) \). Thus, the probability that the bin exceeds twice its expectation is \( \Pr[W > 2\mu] = O(1/\mu^2) \).
Our starting point is to observe that, using simple tabulation instead of 4-independence, one obtains the following:

\[
E[(W - \mu)^4] = O \left( \frac{1}{b} \sum_i w_i^4 + \left( \frac{1}{b^2} + \frac{4q}{b^3} \right) \left( \sum_i w_i^2 \right)^2 \right)
\]  

(7)

As long as the number of bins is \( b \gg 1 \), and the number of characters \( q = O(1) \), the bound is effectively equivalent to (6). In other words, simple tabulation is essentially as effective for load balancing as 4-independent functions. The proof is an elementary combinatorial reduction to Cauchy-Schwartz. (We note that a somewhat related bound is shown in recent independent work by [CLM09], which is more useful in streaming contexts.)

However, to get an application to linear probing, this bound does not suffice. The point is that the query plays a special role: we are not interested in the 4th moment of bin 0, but the 4th moment of the bin containing the query. For this reason, [PPR09] required 5-independence for the analysis of linear probing. As we show in our lower bounds, this is not an artifact of the analysis: with 4-independence, the hash code of the query may be correlated with the location of loaded bins, and this gives an \( \Omega(\log n) \) bound on the expected query time.

To handle this, we prove a surprising result: among any 5 keys, at least one has a hash code independent of the rest. \( \text{(Which one is independent depends on the particular keys.)} \) This proof diverges from all previous analyses of tabulation hashing, in that it is not a peeling argument. We then demonstrate that this 1-in-5 property is enough to recover bound (7), where \( W \) is now the weight in the query’s bin.

Plugging (7) into the analysis of [PPR09], we immediately obtain constant expected time per operation. When dealing with more complicated universes such as variable length strings, a later analysis of [Tho09] proposed a two-stage hash function obtained as \( h_5 \circ h_2 \), where \( h_2 \) is merely a universal hash function to a domain of size \( O(n) \), and \( h_5 \) is a 5-independent hash function from \( O(n) \) to itself. Our result implies that simple tabulation suffices as a replacement for \( h_5 \), since the analysis of [Tho09] also builds on a 4th moment bound.

E.1 Review of the Standard Argument

First we will recall the standard proof of the 4th moment bound (6) assuming 5-independence as it is done in [KR93, PPR09, Tho09]. Next we discuss how to get our bound (7) for our simple tabulation. Using the notations from the previous section, we further define \( Y_i = X_i - p_i \), noting that \( E[Y_i] = 0 \). Now

\[
E[(W - \mu)^4] = \sum_{(i_1, i_2, i_3, i_4) \in S^4} E[w_{i_1} Y_{i_1} w_{i_2} Y_{i_2} w_{i_3} Y_{i_3} w_{i_4} Y_{i_4}].
\]  

(8)

Assuming that the \( h \) is 5-independent, we have that all distinct variables in the terms are independent of each other and of the query key. Therefore the expectation of a term is zero if some key appears only once. This leaves us with terms involving either only one key, or each of two keys twice. Thus (6) follows.

Now, in the case of our simple tabulation, we may have that distinct variables are dependent. However, we know that (6) bounds the sum of all terms in (8) where the distinct variables are independent.

E.2 Dependence

To prove (7), we need to bound the sum of the terms in (8) where distinct variables depend on each other or the query key. We will now discuss the different types of dependence. We are considering a set \( X \) of up to 5 distinct keys, and we think of them as placed in a matrix whose rows are the keys and whose columns are the characters from the same position within the keys. As a general tool, we use [Sie04, Lemma 2.6]:

**Lemma 20** (Siegel’s pealing lemma). \( \text{If in a set } X \text{ of keys, we have a key } x \text{ with a character that is unique in its column, then its hash value } h(x) \text{ is independent of the hash value of the other keys in } X. \)
Actually, Siegel considers some extra derived characters, but for our simple tabulation, there are no
derived characters. Trivially, with two or three keys, each column contains a unique character, so the hash
values of up to three keys are always independent. With 4 keys we get dependence if and only if each
column has either just one character, or two characters, each appearing twice. In the dependent case, the xor
of all the 4 keys is 0. Here we will show

**Theorem 21.** With the simple tabulation, given any 5 distinct keys, there will be one whose hash value is
independent of the other 4 hash values.

With 5 keys, there may be no unique characters, so Theorem 21 does not follow from Siegel’s peeling
lemma. As a side note, we observe that Theorem 21 essentially implies that any 4-independent tabulation
based scheme is also 5-independent, and in particular, this holds for the 4-independent scheme from [TZ04]
which also tabulated some special derived characters. The previous proof of 5-independence for this scheme
[TZ09] was more complicated and used special properties of the derived characters.

With 5 distinct keys, we known from Theorem 21 that one is independent. If the independent one is a
stored key, we get a zero factor, so we only get a contribution if it is the query key that is independent.

The other case to consider is with 4 distinct keys. Recall that the query key is always distinct. In this
case we therefore have two terms from the same key.

### E.3 Among Five Keys, One Hashes Independently

We will now prove the Theorem 21, that if we have 5 keys, then simple tabulation hashes one of them
independently. Assume some fixing $X$ of the 5 keys. We think of these as forming a $5 \times q$ matrix, sometimes
referring to the keys as rows numbered 1,...,5. We define the hashing over a subset $C \subseteq [q]$ of the columns
as $h_C(x) = \bigoplus_{i \in C} h_i(x_i)$.

**Fact 22.** If a key $x \in X$ hashes independently over a subset of the columns, the full hashing of the row is
independent.

This fact is the bases of the Siegel’s peeling lemma, where we pick out a single column where a key
has a unique character that is hashed independently. We shall use it more generally considering multiple
columns. We need one more useful fact.

**Fact 23.** Consider a subset $C$ of the columns, a key $x$ and a subset $Y \subset X$ not involving $x$. Suppose every
remaining key $f \not\in Y \cup \{x\}$ has a hash value $h_C(f)$ which is a function of the hash values $\{h_C(y), y \in Y\}$. If $h_C(x)$ is independent of $\{h_C(y), y \in Y\}$, then $h_C(x)$ is independent of all the other hash values.

Trivially we can discard all columns with only a single character, and by Siegel’s peeling lemma, we can
assume no character is unique in its column. This means that each column has a “minor” character appearing
once and a “major” character appearing thrice. As a signature of the column, we replace the minor with a 0
and the major with a 1. We note that if two columns have the same signature, then the hashing has the same
distribution if we remove one of these columns. Hence we can assume that each signature appears only once
and that each signature has three 1s and two 0s.

**Lemma 24.** If two columns have non-overlapping minors, then one hash value is independent.

**Proof.** W.l.o.g. the signatures of the two columns $C$ look like.

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

We will can now apply Fact 23 with $Y = \{1, 3\}$ and $x = 5$. Trivially $h_C(2) = h_C(1)$ and $h_C(4) = h_C(2)$. When restricted to rows $\{1, 3, 5\}$, we see that rows 1 and 3 each has a unique character, so independence follows by peeling.
We are now ready to prove Theorem 21. Up to a permutation of the rows, we will argue that we have three columns of the following form:

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

The first column is just a generic signature. Since the full rows are distinct, there can be at most one row that is all 1s. Therefore we can find a row with a 0 where the first column had a 1. By Lemma 24, the new column has a 0 shared with the previous column. We repeat this argument getting a third column which has a new 0 shared with both the previous columns. We now note that $h_C(1) = h_C(2) \oplus h_C(3) \oplus h_C(4)$. Hence by Fact 23 it suffices to prove that rows 2, ..., 5 are independent. Now rows 2, ..., 4 have a unique character, so the result follows by peeling. This completes the proof of Theorem 21.

### E.4 Contribution When All Keys are Distinct

We will now bound the contribution to

$$E[(W - \mu)^4] = \sum_{(i_1, i_2, i_3, i_4) \in S^4} E[w_{i_1} Y_1 w_{i_2} Y_2 w_{i_3} Y_3 w_{i_4} Y_4]$$

from terms where all keys are distinct. The query key is distinct by definition. We wish prove

$$\sum_{\text{distinct } i_1, i_2, i_3, i_4 \in S^4} E[w_{i_1} Y_1 w_{i_2} Y_2 w_{i_3} Y_3 w_{i_4} Y_4] = O \left( 4^3 \bar{p}^3 \left( \sum_{i \in S} w_i^2 \right)^2 \right). \quad (9)$$

We will often identify a key with its hash value, so when we say that keys are dependent, we mean that their hash values are dependent. If one of the stored keys is independent, we have a zero factor, hence no contribution. From Theorem 21 we know that some key is independent, so we can assume that this is the query key. Thus, we can think of the query set $Q$ as fixed first. The four stored keys are independent of $Q$ but depend on each other. Despite this dependence, we know that any three of them are independent, so the probability that $X_{i_j} = 1$ is $\bar{p}^3$. Hence this factor in the bound. More formally, for distinct dependent keys $i_1, i_2, i_3, i_4$, we have the random factor $Y_1 Y_2 Y_3 Y_4 = (X_{i_1} - p_{i-1})(X_{i_2} - p_{i-2})(X_{i_3} - p_{i-3})(X_{i_4} - p_{i-4})$ which is only positive if we have an even number of $X_{i_j}$ that are 1. The case of all 1s happened with probability $\bar{p}^3$ and has value below 1. The probability that any two given $X_{i_j}$ are 1 is bounded by $\bar{p}^2$, and if the other two are zero, the product value is less than $\bar{p}^2$. Hence, for any distinct keys $i_1, i_2, i_3, i_4$, we have $E[Y_1 Y_2 Y_3 Y_4] = O(\bar{p}^3)$. Using this common bound for the expected value of the random factors, it remains to prove

$$\sum_{\text{distinct dependent } i_1, i_2, i_3, i_4 \in S} w_{i_1} w_{i_2} w_{i_3} w_{i_4} = O \left( 4^3 \left( \sum_{i \in S} w_i^2 \right)^2 \right). \quad (10)$$

Consider the 4 keys $i_1, ..., i_4$ represented as rows 1, ..., 4. We divide the columns into 4 types, $j = 1, ..., 4$. For type 1, all rows have the same character. For type $j = 2, 3, 4$, row $j$ has the same character as row 1 while the two other rows share a different character. We have $4^3$ ways of distributing types on columns, hence the factor $4^3$ in (10). We now fix a combination of column types. We say the a quadruple of keys $(i_1, ..., i_4)$ is “type valid” if it respect this types. We will prove

$$\sum_{\text{type valid } (i_1, i_2, i_3, i_4) \in S^4} w_{i_1} w_{i_2} w_{i_3} w_{i_4} \leq \left( \sum_{i \in S} w_i^2 \right)^2. \quad (11)$$

We first partition the keys based on the characters in the type 2 columns. We want $i_1$ and $i_2$ to be in one set and $i_3$ and $i_4$ to be in another set. Let $S_1, ..., S_m$ be the sets of the partition. For any $S_a$ and $S_b$, we note that
for each pair \((i_1, i_2) \in S_a\), there can be at most one pair \((i_3, i_4) \in S_b\) leading to the right type combination. The point is that \(i_1\) and \(i_2\) contain all the relevant characters in the columns that are not type 2, so when we know \(S_a\) and \(S_b\), \(i_1\) and \(i_2\), we know exactly how keys \(i_3\) and \(i_4\) look if they exist among the stored keys.

It follows that we can make an enumeration of the pairs \((i_1, i_2) \in S_a\) and the pairs \((i_3, i_4) \in S_b\) such that \(i_1, i_2, i_3, i_4\) can only be a valid type combination if \((i_1, i_2)\) and \((i_3, i_4)\) have the same number in their respective enumerations. Applying Cauchy-Schwartz as the first step, we get

\[
\sum_{\text{type valid } (i_1,i_2,i_3,i_4), \ i_1,i_2 \in S_a, \ i_3,i_4 \in S_b} w_{i_1} w_{i_2} w_{i_3} w_{i_4} \leq \sqrt{\left( \sum_{i_1,i_2 \in S_a} w_{i_1} w_{i_2} \right)^2 \left( \sum_{i_3,i_4 \in S_b} w_{i_3} w_{i_4} \right)^2} = \sum_{i \in S_a} w_i^2 \sum_{j \in S_b} w_j^2
\]

Now we are basically done because

\[
\sum_{\text{type valid } (i_1,i_2,i_3,i_4) \in S^4} w_{i_1} w_{i_2} w_{i_3} w_{i_4} = \sum_{a,b \in \{1,\ldots,m\}} \left( \sum_{\text{type valid } (i_1,i_2,i_3,i_4), \ i_1,i_2 \in S_a, \ i_3,i_4 \in S_b} w_{i_1} w_{i_2} w_{i_3} w_{i_4} \right)
\leq \sum_{a,b \in \{1,\ldots,m\}} \left( \sum_{i \in S_a} w_i^2 \sum_{j \in S_b} w_j^2 \right) = \left( \sum_{i \in S} w_i^2 \right)^2
\]

This completes the proof of (11), hence of (10) and (9).

### E.5 Four Distinct Keys, Including the Query

We now consider the case where we have only 4 distinct keys including the query key. This means that we are considering terms in which one key appears twice, and the rest only once. Thus we are considering terms of the form

\[
w_i^2 w_j w_k \mathbb{E}[Y_i^2 Y_j Y_k]
\]

The term is only non-zero if we have dependence, and then we know that any key is uniquely derived from the three other keys. Assume \(w_j \geq w_k\). We know that \(Y_i\) and \(Y_j\) can be picked independently of the query, but then \(Y_k\) depends on the other choices. Expecting the worst-case, we have \(X_j = X_k\), and then

\[
w_i^2 w_j w_k \mathbb{E}[Y_i^2 Y_j Y_k] < w_i^2 w_j^2 \mathbb{E}[X_i^2 X_j^2] \leq \tilde{p}^2 w_i^2 w_j^2
\]

The next question is how many times we see the index combination \(i^2, j, k\) among all the quadruples \((i_1, i_2, i_3, i_4)\). We know that there is at most one dependent \(k\) given \(i, j, k\), and in fact, we only count this \(k\) if \(w_j \geq w_k\). For each value of \(i\), we pick two (out of four) positions for \(i\) and one (out of two remaining) position for \(j\), while the remaining position goes to the only possible value of \(k\) if any. Thus we get given keys \(i\) and \(j\) in at most \(\binom{4}{2} \binom{2}{1} = 12\) ways, so total contribution is bounded by

\[
12 \tilde{p}^2 \sum_{i \neq j} (w_i^2 w_j^2) = O \left( \tilde{p}^2 \left( \sum_i w_i^2 \right)^2 \right).
\]

This actually fits within our general bound (6) with full 5-independence. Together with (9), this completes the proof of (7).