# Cache-Oblivious Hashing*

Rasmus Pagh
IT University of Copenhagen
Copenhagen, Denmark
pagh@itu.dk

Zhewei Wei Ke Yi Qin Zhang
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong, China
{wzxac, yike, qinzhang}@cse.ust.hk

## ABSTRACT

The hash table, especially its external memory version, is one of the most important index structures in large databases. Assuming a truly random hash function, it is known that in a standard external hash table with block size $b$, searching for a particular key only takes expected average $t_q = 1+1/2^{\Omega(b)}$ disk accesses for any load factor $\alpha$ bounded away from 1. However, such near-perfect performance is achieved only when $b$ is known and the hash table is particularly tuned for working with such a blocking. In this paper we study if it is possible to build a cache-oblivious hash table that works well with any blocking. Such a hash table will automatically perform well across all levels of the memory hierarchy and does not need any hardware-specific tuning, an important feature in autonomous databases.

We first show that linear probing, a classical collision resolution strategy for hash tables, can be easily made cache-oblivious but it only achieves $t_q = 1 + O(\alpha/b)$. Then we demonstrate that it is possible to obtain $t_q = 1 + 1/2^{\Omega(b)}$, thus matching the cache-aware bound, if the following two conditions hold: (a) $b$ is a power of 2; and (b) every block starts at a memory address divisible by $b$. Both conditions hold on a real machine, although they are not stated in the cache-oblivious model. Interestingly, we also show that neither condition is dispensable: if either of them is removed, the best obtainable bound is $t_q = 1 + O(\alpha/b)$, which is exactly what linear probing achieves.

*The work of Rasmus Pagh was supported by the Danish National Research Foundation, as part of the project "Scalable Query Evaluation in Relational Database Systems". The work of Zhewei Wei, Ke Yi, and Qin Zhang was supported by a Hong Kong DAG grant (DAG07/08). Qin Zhang was in addition supported by Hong Kong CERG Grant 613507.


*PODS'10,* June 6–11, 2010, Indianapolis, Indiana, USA.


## Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems; E.2.4 [**Data**]: Data storage representations—*hash-table representations*

## General Terms

Algorithms, theory

## Keywords

Cache-oblivious algorithms, hashing

## 1. INTRODUCTION

The hash table is one of the most fundamental index structures in databases. It stores a set of $n$ keys from a universe $[u]$ in linear space, while allowing us to search for any particular key efficiently. It is also one of the simplest data structures: Let $h : [u] \to [r]$ be a hash function. The table has size $r \geq n$ and we simply store key $x$ in position $h(x)$. If that position already contains some other key, one can use various collision resolution strategies, among which *chaining* and *linear probing* are the most common ones. In chaining, we simply store all keys that are mapped to same position in a list associated with that position. In linear probing, if position $h(x)$ is already occupied when $x$ is being inserted, we successively probe positions $h(x), h(x)+1, \ldots, r-1, 0, 1, \ldots, h(x)-1$ until an empty position is found and we will put $x$ there. To do a search on $x$, we follow the same probing sequence, until $x$ is found or an empty position is encountered, in which case we know that $x$ is not stored in the table. It is known that linear probing generally outperforms chaining in practice due to its sequential access pattern, provided that the *load factor* $\alpha = n/r$ is not too close to 1.

The mathematical analysis of hashing is usually considered as the birth of analysis of algorithms [14], and it is still attracting a lot of attention nowadays. Most analyses on hashing assume $h$ to be a truly random function, i.e., each $h(x)$ is independently uniformly distributed on $[r]$. It has been observed that these analyses match what actually happens on real-world data surprisingly well, even with some very simple hash functions. Recently, some theoretical explanation [18] has also been put forward justifying such an assumption. We will also adopt the truly random hash function assumption in this paper. Under such an assumption, Knuth [14] showed that the expected average number of probes during a search using linear probing is (averaged

over all keys):

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \quad \text{(successful search)};$$
$$C'_n \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) \quad \text{(unsuccessful search)}.$$

Thus, for a typical load factor $\alpha = 0.7$, we expect to make 2.17 probes if the searched key is in the table, and 6.05 probes if it is not.

In large databases, the hash table is usually stored in external memory, and data is accessed in terms of blocks. In this setting, we care about the number of blocks accessed (I/Os) when performing a search. The number of I/Os is clearly at most the number of probes, but such a naive analysis is too pessimistic. Interestingly, Knuth [14] showed that the external version of linear probing has a search cost of $1 + 1/2^{\Omega(b)}$ I/Os (for both successful and unsuccessful searches), where $b$ is the block size. Here and further we assume that the load factor $\alpha$ is bounded away from 1. In the external version of linear probing, the table consists of $r/b$ blocks, and correspondingly we use a hash function $h : [u] \to [r/b]$. To do a search on $x$, we successively access blocks $h(x), h(x)+1, \ldots$ until $x$ is found or a non-full block is encountered. The intuitive explanation for this extremely close-to-one cost is that since a block has size $b$, we will not have a collision unless more than $b$ keys are hashed into this block, which happens with probability exponentially small in $b$. Knuth [14] actually derived the constant in the big-Omega, showing that for reasonably large $b$ (larger than 10), the number of I/Os is very close to 1, much smaller than the number of probes, demonstrating the excellent locality of external linear probing. Meanwhile, a natural external version of chaining also achieves the same bound. These results basically have explained why hash tables work so well in external memory.

These classical analyses assumed a simple two-level memory model [2], where the (sufficiently large) external memory is partitioned into blocks of size $b$ and are fetched into the internal memory of size $m$ as they are probed. Here both sizes are measured in terms of $(\log u)$-bit words. Starting in the late 90's tremendous efforts have been devoted to the design and analysis of data structures that work well not only in a two-level memory model, but also in a memory hierarchy that consists of any number of levels, where each level has a different capacity $m$ and block size $b$. Among them, the most successful approach is the *cache-oblivious* model [10] due to its elegance and simplicity. This model actually only features two levels of memory: a data structure is laid out in external memory and accessed in exactly the same way as in the standard two-level model, but the additional requirement is that the structure is unaware of the block size $b$, or equivalently, the structure is laid out in external memory in a way that works for all block sizes[1]. Thus a cache-oblivious data structure automatically works in a memory hierarchy. More precisely, if we can show that the cost of some operation on a cache-oblivious structure is $f(n, b)$ I/Os in the two-level model, then the number of block transfers will always be $f(n, b)$ between any two levels in a memory hierarchy with multiple levels, where the $b$ simply becomes the block size of that corresponding level. Another major benefit of cache-oblivious algorithms and data structures is that they achieve their guaranteed performance without any hardware-specific tuning. This is particularly important in autonomous databases, and is in fact the main motivation of the recent efforts in bringing cache-oblivious techniques to databases, such as EaseDB [12].

Note that the external versions of linear probing and chaining mentioned above only work for a single $b$, so they are not cache-oblivious. In this paper we investigate whether it is possible to lay out a hash table such that its search cost matches its cache-aware version, i.e., $1 + 1/2^{\Omega(b)}$ I/Os, for all block sizes $b$.

### *Our results.*

A straightforward way of making the hash table cache-oblivious is to simply use linear probing but ignoring the blocking at all[2]. One would expect it to work well irrespective of the block size since it uses only sequential probes. However, in Section 2 we show that its search cost is $1 + O(\alpha/b)$ I/Os. In fact, we also derive the constant in the big-Oh, which depends on $C_n$ and $C'_n$. This is worse than its cache-aware version that is particularly tuned to work with a single $b$. The gap is in some sense exponential, if we are concerned with the fraction of keys that cannot be found with a single cache miss (note that an average search cost of $t_q = 1 + \varepsilon$ means that at most a fraction of $\varepsilon$ keys need two or more I/Os).

Next, we explore other collision resolution strategies to see if they work better in the cache-oblivious model. In Section 3, we show that the *blocked probing* algorithm [20] achieves the desired $1 + 1/2^{\Omega(b)}$ search cost, but under the following two conditions: (a) $b$ is a power of 2; and (b) every block starts at a memory address divisible by $b$. Neither of these conditions is stated in the cache-oblivious model, but they indeed hold on all real machines. This raises the theoretical question of whether $1 + 1/2^{\Omega(b)}$ is achievable in the "true" cache-oblivious model. In Section 4 we show that neither condition is dispensable. Specifically, we prove that if the hash table is only required to work for a single $b$ but an arbitrary shift of the layout, or if (b) holds but the hash table is required to work for all $b$, then the best obtainable search cost is $1 + O(\alpha/b)$ I/Os, which exactly matches what linear probing achieves. Our lower bound model puts no restrictions on the structure of the hash table, except that each key is treated as an atomic element, known as the *indivisibility* assumption.

### *Related results.*

Hashing is perhaps one of the most studied problems in computer science. Most works on hashing assume a truly random function, as we do in this paper. Since such a function requires a large space to describe, there are also a lot of works on hashing using explicit and efficient hash functions [6, 20]. Meanwhile, although most works focus on the expected search cost, there are also hashing schemes that guarantee good worst-case search costs [9, 21]. Hashing has been well studied in the external memory model. The $1 + 1/2^{\Omega(b)}$ search cost holds as long as the load factor $\alpha$

[1]Strictly speaking the structure should be unaware of both $m$ and $b$. But for most data structure problems the operations on the structure are always oblivious to $m$, so we only need to require that the layout works for all $b$.

[2]Chaining would perform worse cache-obliviously because the list associated with each position is not laid out consecutively.

is bounded away from 1 [14], and there are various techniques in the database literature to keep the load factor in a desired range, such as *extensible hashing* [8] or *linear hashing* [17]. Jensen and Pagh [13] designed a hashing scheme that has $\alpha = 1 - O(1/\sqrt{b})$ while supporting searches with $1 + O(1/\sqrt{b})$ I/Os. In all these hashing schemes a small faction of the keys still need two or more disk accesses to retrieve. Meanwhile there are also schemes that guarantee a single I/O to retrieve any key [11, 16], but they all need the internal memory to have size $m = \Theta(n/b)$. Note that on the other hand, all the other hashing schemes achieving $t_q = 1+\varepsilon$ only need the internal memory to store a constant number of blocks.

The cache-oblivious model was proposed by Frigo et al. [10], which introduces a clean and elegant way to modeling memory hierarchies. Previous approaches attempted to model a memory hierarchy directly, but did not have much success due to the complicated models. Since then, cache-oblivious algorithms and data structures have received a lot of attention, and most fundamental problems have been solved. For example, cache-oblivious sorting takes $O(\frac{n}{b}\log_{m/b}\frac{n}{b})$ I/Os [10], and a cache-oblivious B-tree takes $O(\log_b n)$ I/Os for a search [4]. Please see the survey [7] for other results. However, hashing has not been considered in the cache-oblivious model so far. In most cases the cache-oblivious bounds match their cache-aware versions, and it has always be an interesting problem to see for what problems we have a separation between the cache-oblivious model and the cache-aware model. Until today there have been only three separation results [1, 3, 5]. Our lower bound adds to that list, furthering our understanding of cache-obliviousness.

## 2. ANALYSIS OF LINEAR PROBING IN THE CACHE-OBLIVIOUS MODEL

Linear probing while ignoring the blocking is naturally cache-oblivious. In this section we analyze its search I/O cost, which turns out to delicately depend on $C_n$ and $C'_n$, the expected number of probes in a successful and unsuccessful search, respectively. Note that the equalities in the theorem below are exact, though we only know the asymptotic formulas for $C_n$ and $C'_n$.

THEOREM 1. *Let $CO_n$ and $CO'_n$ denote the expected number of I/Os for a successful and an unsuccessful search, respectively. For any block size $b$, we have*

$$CO_n = 1 + (C_n - 1)/b$$
$$CO'_n = 1 + (C'_n - 1)/b.$$

PROOF. Let $r$ be the size of the hash table, which is divided into $r/b$ blocks $B_0, \ldots, B_{r/b-1}$ (assuming that $r$ is a multiple of $b$ for simplicity). The block $B_l$ spans positions $lb, lb+1, \ldots, lb+b-1$. We will first consider unsuccessful searches. Define $p(i,j), i \neq j$, to be the probability that the hash table has position $j$ empty, while positions $i$ through $j-1$ occupied (wrapping around when necessary), and $p(i,i)$ the probability of position $i$ being empty. Note that the number of occupied positions is $n$, so $p(i,j) = 0$ for any $j \notin \{i, i+1, \ldots, i+n\}$ (wrapping around when necessary). By the circular symmetry of linear probing and the truly random hash function assumption, $p(0,k)$ is exactly the probability that an unsuccessful search of a key $x$ takes exactly $k+1$ probes. Thus we have [14]:

$$C'_n = \sum_{k=0}^{n}(k+1)p(0,k). \tag{1}$$

Let $p_k$ be the probability that an unsuccessful search takes $k+1$ I/Os. Below we will relate $p_k$ with the $p(0,k)$'s. Note that for a search to cost $k+1$ I/Os, the probe sequence will visit $k+1$ consecutive blocks, that is, it will probe the positions $lb+i, lb+i+1, \ldots, (l+k)b+j$, for any $0 \le i, j \le b-1$ if $k \ge 1$ (wrapping around when necessary), or any $0 \le i \le j \le b-1$ if $k = 0$. For an unsuccessful search, positions $lb+i$ through $(l+k)b+j-1$ must be occupied and position $(l+k)b+j$ must be empty. Since $h(x)$ is uniformly distributed in $[r]$, each of the $r$ positions in the table is equally likely to be the starting position. For $\alpha$ bounded away from 1, we have $r > n+b$ for $n$ sufficiently large and thus $p(lb+i, lb+j) = 0$ for $i > j$. So we can write $p_k$ for any $k$ as

$$\begin{aligned}
p_k &= \sum_{l=0}^{r/b-1}\sum_{i=0}^{b-1}\Pr[h(x) = lb+i]\sum_{j=0}^{b-1}p(lb+i,(l+k)b+j)\\
&= \sum_{l=0}^{r/b-1}\sum_{i=0}^{b-1}\frac{1}{r}\sum_{j=0}^{b-1}p\left(0,kb+j-i\right) \text{ (by circular symmetry)}\\
&= \frac{1}{b}\sum_{i=0}^{b-1}\sum_{j=0}^{b-1}p(0,kb+j-i).
\end{aligned}$$

The last formula can be divided into two parts: the summation of all $p(0,kb-b+s)$ for $s = 0, \ldots, b-1$, and the summation of all $p(0,kb+s)$ for $s = 0, \ldots, b-1$. Note that each $p(0,kb-b+s)$ contributes $s$ times and each $p(0,kb+s)$ contributes $b-s$ times in the summation. Thus $p_k$ can be expressed as

$$p_k = \frac{1}{b}\left(\sum_{s=0}^{b-1}sp(0,kb-b+s) + \sum_{s=0}^{b-1}(b-s)p(0,kb+s)\right).$$

Now we can compute $CO'_n$ as follows:

$$\begin{aligned}
CO'_n &= \sum_{k=0}^{\lfloor n/b\rfloor+1}(k+1)p_k\\
&= \sum_{k=0}^{\lfloor n/b\rfloor+1}\sum_{s=0}^{b-1}\frac{k+1}{b}\left(\sum_{s=0}^{b-1}sp(0,kb-b+s)\right.\\
&\quad\left.+\sum_{s=0}^{b-1}(b-s)p(0,kb+s)\right)\\
&= \frac{1}{b}\left(\sum_{k=-1}^{\lfloor n/b\rfloor}\sum_{s=0}^{b-1}(k+2)sp(0,kb+s)\right.\\
&\quad\left.+\sum_{k=0}^{\lfloor n/b\rfloor1}\sum_{s=0}^{b-1}(k+1)(b-s)p(0,kb+s)\right).
\end{aligned}$$

In the last equality we substitute the index $k$ with $k-1$ for the first summation. Noting that $k = -1$ or $k = \lfloor n/b\rfloor + 1$ implies $p(0,kb+s) = 0$ regardless what $s$ is, we can simplify

the equation as

$$\begin{aligned} CO'_n &= \frac{1}{b}\left(\sum_{k=0}^{\lfloor n/b\rfloor}\sum_{s=0}^{b-1}[(k+2)s+(k+1)(b-s)]\,p(0,kb+s)\right) \\ &= \frac{1}{b}\left(\sum_{k=0}^{\lfloor n/b\rfloor}\sum_{s=0}^{b-1}(kb+b+s)p(0,kb+s)\right) \\ &= \frac{1}{b}\sum_{t=0}^{n}(t+b)p(0,t) \\ &= 1-\frac{1}{b}+\sum_{t=0}^{n}(t+1)p(0,t) \\ &= 1-\frac{1}{b}+\frac{C'_n}{b}. \end{aligned}$$

For the expected successful query cost $CO_n$, we have:

$$CO_n = \frac{1}{n}\sum_{k=0}^{n-1} CO'_k = 1-\frac{1}{b}+\frac{\sum_{k=0}^{n-1} C'_k}{nb} = 1-\frac{1}{b}+\frac{C_n}{b}.$$

□

Combining Knuth's results with our analysis, we can conclude that the I/O cost of directly applying linear probing in the cache-oblivious model is $1+\Theta(\alpha/b)$, which is a lot worse than its the external version that is aware of the blocking.

# 3. BLOCKED PROBING

Standard linear probing maintains the invariant that each key $x$ is placed as close as possible to position $h(x)$ in the probe sequence, in the sense that no single key can be moved to decrease the length of a probe sequence. *Blocked probing* is a variant of linear probing proposed by Pagh, Pagh and Ružić [20], who used it to derive optimal performance (as a function of the load factor $\alpha$) assuming only 5-wise independent hash functions.

In this paper we are concerned with strong bounds on the expected number of probes, for which the probability space of 5-wise independent hash functions is not sufficiently large: If the probability of using more than 1 I/O should be $2^{-\Omega(b)}$, we need a probability space of size at least $2^{\Omega(b)}$. More generally, any explicit family of hash functions that allows performance that tends to 1 I/O as $b$ increases must use space that depends on $b$. Thus, there is no hope to make such a hash function cache-oblivious. Instead, as in most works on hashing, we perform our analysis assuming a truly random hash function. All experience shows that analyses performed under this assumption closely match observed performance when using high-quality heuristic hash functions.

## 3.1 Algorithm description

As described in [20], blocked probing assumes a hash table whose size $r$ is a power of two. It also assumes that $r$ is fixed, i.e., there is no notion of dynamically adjusting the capacity of the hash table; at the end of this section we sketch how to handle the general case. Suppose that the key $x$ is stored in location $i_x$. Following [20] we define the distance measure $d(x,i_x)$ to be equal to the position of the most significant bit in which $h(x)$ and $i_x$ differ (the least significant bit is said to be at position 1), and $d(x,i_x)=0$ in case $i_x=h(x)$. Let $I(x,j)=\{i \mid d(x,i)\le j\}$. Note that $I(x,j)$ is the *aligned* block of size $2^j$ that contains $h(x)$. The invariant of blocked probing is that each key is stored as close as possible to $h(x)$ in the sense that $i_x\in I(x,j)$ if there is sufficient space, i.e., if the number of keys with hash values in $I(x,j)$ is at most $|I(x,j)|=2^j$. A thorough discussion of the operations of blocked probing can be found in [20], but we sketch them here for completeness.

When *inserting* a key $x$ the invariant is maintained by searching, for $j=0,1,2,\ldots$, for a location $i\in I(x,j)$ where $x$ could be placed. For each $j$ we first check if there is an empty location in $I(x,j)$ and put $x$ there if there is one. Otherwise, we look for a location $i_{x'}\in I(x,j)$ that contains a key $x'$ with $d(x',i_{x'})>j$ (implying that $h(x')\notin I(x,j)$). If there is such an $x'$ we swap $x$ and $x'$, and continue the insertion process with $x'$. If both attempts fail we move on to the next $j$.

A *search* for $x$ proceeds by inspecting, for $j=0,1,2,\ldots$, the locations of $I(x,j)$ until either $x$ is found, or we do not find $x$ but find instead an empty location or a key $x'$ with $d(x',i_{x'})>j$. In the latter cases, the invariant tells us that $x$ is not present in the hash table.

*Deletion* of a key $x\in I(x,j)\backslash I(x,j-1)$ needs to check if there is a key stored in $I(x,j+1)\backslash I(x,j)$ that could be stored in $I(x,j)$ — if this is the case it is copied to the empty location, and the old copy is deleted recursively.

## 3.2 Cache-oblivious analysis of blocked probing

Let $S$ denote the set of keys involved in a given operation (insertion, deletion, successful or unsuccessful search), including the key $x$ specified by the query or update ($x$ may or may not be in the hash table). Define $X_j$ as the number of keys in $S$ with hash value in the aligned block of size $2^j$ containing $h(x)$, i.e., $X_j=|\{y\in S \mid h(y)\in I(x,j)\}|$. As in [20] we observe that we will not visit any locations outside of $I(x,j^*)$, where $j^*=\min\{j \mid X_j\le 2^j\}$. We know that $h(x)\in I(x,j)$, but for any key $y\neq x$ we have, by the full randomness assumption, that $\Pr[h(y)\in I(x,j)]=2^j/r$. This means that $X_j-1$ follows a binomial distribution with expectation bounded by $n2^j/r=\alpha 2^j$. By Chernoff bounds $\Pr[X_j-1>2^j-1]\le 2^{-(1-\alpha)^2\,(2^j-1)/2}$, so the probability that $j^*>j$ is at most $2^{-(1-\alpha)^2\,(2^j-1)/2}$.

By the assumption that $b$ is a power of 2 and storage blocks are aligned to multiples of $b$, we have that all locations in $I(x,\log b)$ can be visited in 1 I/O. More generally, if the search goes on to step $j^*>\log b$ the number of I/Os required is $2^{j^*}/b$, since $I(x,j^*)$ consists of that many blocks. To compute the expected number of blocks involved in an operation, in addition to the first I/O that retrieves $I(x,\log b)$, we sum over all possible values of $j^*>\log b$ the cost $2^{j^*}/b$ multiplied by the probability that $j^*$ steps or more is used:

$$1+\sum_{j=1+\log b}^{\infty}(2^j/b)\,2^{-(1-\alpha)^2\,(2^j-1)/2} = 1+2^{-\Omega((1-\alpha)^2\,b)},$$

The upper bound uses the fact that the sum is rapidly decreasing as $j$ increases, and hence is dominated by the first term. In conclusion, we have upper bounded the expected I/O cost for a search, insertion, or deletion, to $1+2^{-\Omega((1-\alpha)^2\,b)}$, which is $1+1/2^{\Omega(b)}$ for $\alpha$ bounded away from 1.

## 3.3 Cache-oblivious dynamic hash tables

The standard doubling/halving strategy can be used to

maintain the load factor $\alpha$ in the range $1/2-\varepsilon/2 \le \alpha \le 1-\varepsilon$ as we insert and delete keys in the hash table where $\varepsilon > 0$ is any small constant. In such a range the expected I/O cost per operation is $1 + 1/2^{\Omega(b)}$ I/Os using the blocked probing scheme described above. In particular, we always use a hash table of size $r$ that is a power of 2. Let $g : [u] \to [u]$ be a "mother" hash function. When the table's size is $r$, we take the $\log r$ least significant bits of $g(x)$ as $h(x)$. When $\alpha = n/r$ goes beyond the range $[1/2 - \varepsilon/2, 1 - \varepsilon]$ we double or halve $r$ accordingly. This can be done in a simple scan of the hash table in amortized $O(1/b)$ I/Os per key, by simply inserting keys in the order they occur in the table. The analysis uses the fact that the keys to be inserted in a block in the resized hash table are (w.h.p.) in at most two blocks in the original hash table. We omit the rather standard analysis.

However, the above solution has a poor space utilization. A number of methods have been proposed that maintain a higher load factor, and also allow the rehashing to be done incrementally; see [15] for an overview. To our best knowledge these methods are all cache-aware — however, we now describe how they can be made cache-oblivious while maintaining the load factor of $\alpha = 1 - \Theta(\varepsilon)$. Suppose initially $r$ is a power of 2 and $n > (1 - 2\varepsilon)r$. Adjust $\varepsilon$ so that $\varepsilon r$ is also a power of 2; this will not change $\varepsilon$ by more than a factor of 2. The idea is to split the hash table into $1/\varepsilon$ parts using hashing (say, by looking at the first $\log(1/\varepsilon)$ bits of the mother hash function), where each part is handled by a cache-oblivious hash table of size $\varepsilon r$ which stores at most $(1 - \varepsilon)\varepsilon r$ keys. As $n$ changes, the number of parts also changes to maintain the overall load factor at $\alpha = 1 - \Theta(\varepsilon)$. Now this situation is analogous to a standard cache-aware hash table with "block size" being equal to $(1 - \varepsilon)\varepsilon r$, and parts corresponding to blocks. So we may use any cache-aware method that resizes a standard hash table, such as *linear hashing* [17]. These resizing techniques will split or merge parts as needed, and cost is $O(1/b)$ I/Os per insertion/deletion amortized. When $r$ doubles or halves, we rebuild the entire hash table using a new part size $\varepsilon r$. The cache-aware resizing techniques ensures that only $1+1/2^{\Omega(b')}$ parts are accessed upon a query in expectation, where $b'$ is the part size $b' = (1 - \varepsilon)\varepsilon r$. Within each part, our cache-oblivious scheme accesses $1 + 1/2^{\Omega(b)}$ blocks. So as long as $r \gg b$, the overall query cost is still $1 + 1/2^{\Omega(b)}$ I/Os, as desired.

In summary, we can dynamically update our cache-oblivious hash table while maintaining a high load factor. The additional resizing cost is only $O(1/b)$ I/Os amortized.

THEOREM 2. *In the cache-oblivious model where the block size $b$ is a power of 2 and every block starts at a memory address divisible by $b$, there is a dynamic hash table that supports queries in expected average $t_q = 1 + 1/2^{\Omega(b)}$ I/Os, and insertions and deletions of keys in expected amortized $1 + O(1/b)$ I/Os. The load factor can be maintained at $\alpha \ge 1 - \varepsilon$ for any constant $\varepsilon > 0$.*

# 4. LOWER BOUNDS

In this section, we show that the two conditions that the analysis of blocked probing depends upon are both necessary to achieve a $1 + 1/2^{\Omega(b)}$ search cost. Specifically, we prove that when either condition is removed, the best obtainable bound for the expected average cost of a successful search is $1 + O(\alpha/b)$ I/Os. The lower bound proofs do not assume that $\alpha$ is a constant, so it means that we cannot hope to do a lot better even with super-linear space.

## 4.1 The model

Before we present the exact lower bound statements let us first be more precise about our model. Let $U = [u]$ be the universe. The hard input we consider here is a random input in which each key is drawn from $U$ uniformly and independently. Let $I_{\mathbf{u}}$ be such a random input, and $\mathcal{I}$ be the set of all inputs. We will bound from below the expected average cost of a successful search on $I_{\mathbf{u}}$ where the average is taken over all keys in $I_{\mathbf{u}}$. We will only consider deterministic hash tables; the lower bounds also hold for randomized hash tables by invoking *Yao's minimax principle* [19] because we are using a random input. The hash table can employ any hash functions to distribute the input. We assume $u > n^3$, then with probability $1 - O(1/n)$ all keys in $I_{\mathbf{u}}$ are distinct by the birthday paradox.

We assume that all the $n$ keys are stored in a table of size $r$ on external memory[3], possibly with duplication. We model the search algorithm by two functions $f, g : [u] \to [r]$. For any $x \in [u]$, $f(x)$ is the position where the algorithm makes its first probe, while $g(x)$ is the position of the last probe, where key $x$ (or one of its copies) must be located. Note that the internal memory must be able to hold the description of $f$, thus any deterministic hash table can employ a family $\mathcal{F}$ of at most $2^{m \log u}$ such functions. Although the particular $f$ used by the hash table of course can depend on the input $I_{\mathbf{u}}$, the family $\mathcal{F}$ has to be fixed in advance. We do not have any restrictions on $g$, as it is possible for the search algorithm to evaluate $g$ after accessing external memory, except that all $g(x)$'s are distinct for the $n$ keys.

The table is partitioned into blocks of size $b$. For any $x$ such that $f(x) \ne g(x)$, define $g'(x)$ to be $g(x)$ if $f(x) < g(x)$ and $g(x) + 1$ if $f(x) > g(x)$. Then if $g'(x)$ is the first position of a block, at least two blocks must have been accessed, though the reverse is not necessarily true; please refer to Figure 1. For lower bound purposes we will assume optimistically that the search for $x$ needs two I/Os if $g'(x)$ is the first position of a block, and one I/O otherwise. Note that after this abstraction, the search cost is completely characterized by the functions $f, g$ and the blocking.

We will consider the following two blocking models. In the *boundary-oblivious* model, the hash table knows the block size $b$ but not their boundaries. More precisely, how the keys are stored in the table is allowed to depend on $b$, but the layout should work for any shifting $s$, namely when each block spans the positions from $ib - s$ to $(i + 1)b - s - 1$ for $s = 0, 1, \ldots, b - 1$. In the *block-size-oblivious* model, the blocks always start at positions that are multiples of $b$ but the layout is required to work for all $b = 1, \ldots, r$. Below we will show that in either model, the best possible expected average cost of a successful search is $1 + O(\alpha/b)$ I/Os.

## 4.2 Good inputs and bad inputs

For any $I \in \mathcal{I}, f \in \mathcal{F}$, define $\eta_f(I) = \sum_{i \in [r]}(|\{x \in I \mid f(x) = i\}| - 1)$. Intuitively, $\eta_f(I)$ is the number of the overflowed keys; since each position $i$ can only hold one key, at least $\eta_f(I)$ keys in $I$ need a second probe when the hash

[3]Here we do not allow keys to be stored in internal memory: since the memory holds at most $m$ keys, it does not affect the average search cost as long as $n$ is sufficiently larger than $m$.

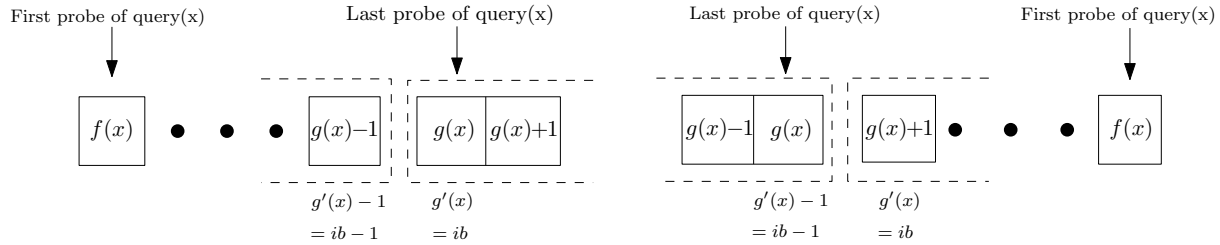


**Figure 1: When two I/Os are needed.**

table uses $f$ to decide its first probe. We say an input $I \in \mathcal{I}$ is *bad* with respect to $f$ if $\eta_f(I) \geq \frac{\alpha}{4}n$, otherwise it is *good*. Let $\mathcal{I}_f$ be the set of all bad inputs with respect to $f$, and $\mathcal{I}_{\mathcal{F}} = \bigcap_{f\in\mathcal{F}} \mathcal{I}_f$ which is the set of inputs that are bad with respect to all $f \in \mathcal{F}$. In our lower bounds we will actually focus only on the bad inputs $\mathcal{I}_{\mathcal{F}}$. The following technical lemma ensures that almost all inputs are in $\mathcal{I}_{\mathcal{F}}$.

LEMMA 1. *For $n > cm\log u/\alpha^2$ where $c$ is some sufficiently large constant and $\alpha = \omega(n^{-1/2})$, $I_{\mathbf{u}}$ is a bad input with respect to all $f \in \mathcal{F}$ with probability $1-o(1)$ as $n \to \infty$.*

The general idea of the proof is the following: We first show that for a particular $f$ and a random $I_{\mathbf{u}}$, the probability that $I_{\mathbf{u}}$ is good with respect to $f$ is $e^{-\Omega(\alpha^2 n)}$. Thus by a union bound, $I_{\mathbf{u}}$ is good for at least one $f \in \mathcal{F}$ with probability at most $e^{-\Omega(\alpha^2 n)} \cdot 2^{m\log u}$. So as long as $n$ is large enough, $I_{\mathbf{u}}$ will be bad with respect to all $f \in \mathcal{F}$ with high probability.

We need the following bin-ball game, which models the way how $f$ works on a uniformly random input:

### *A bin-ball game.*

In a $(n, r, \vec{\beta})$ *bin-ball game*, we throw $n$ balls into $r$ bins independently at random. The probability that a ball goes to the $j$-th bin is $\beta_j$, where $\vec{\beta} = (\beta_0, \ldots, \beta_{r-1})$ is a prefixed distribution. Let $Z$ denote the number of empty bins after $n$ balls are thrown in.

LEMMA 2. *In an $(n, r, \vec{\beta})$ bin-ball game, $\Pr[Z \leq r - n + \frac{\alpha}{4}n] \leq e^{-\Omega(\alpha^2 n)}$, where $\alpha = n/r$.*

PROOF. Note that if $\vec{\beta}$ is the uniform distribution, the problem is known as the *occupancy problem* and the lemma can be proved using properties of martingales [19]. The same proof actually also holds for a nonuniform $\vec{\beta}$, so we just sketch it here:

Let $Z_0$ be the expectation of $Z$ before any ball is thrown in, and let the random variable $Z_i$ be the expectation of $Z$ after the $i$-th ball is thrown in (where the randomness is from the first $i$ balls), for $i = 1, \ldots, n$. Note that $Z_0 = \mathbf{E}[Z]$ and $Z_n = Z$. It can be verified that the sequence $Z_0, Z_1, \ldots, Z_n$ is a martingale, and that $|Z_{i+1} - Z_i| \leq 1$ for all $0 \leq i < n$. Therefore by Azuma's inequality, we get

$$\Pr[Z \leq \mathbf{E}[Z] - \lambda n^{1/2}] \leq 2e^{-\lambda^2/2}.$$

Note that

$$\mathbf{E}[Z] = \sum_{i=0}^{r-1}(1-\beta_i)^n \geq r\left(\frac{r - \sum_{i=0}^{r-1}\beta_i}{r}\right)^n = r\left(1-\frac{1}{r}\right)^n$$
$$\geq r - n + \frac{\alpha}{2}n - \frac{\alpha}{2} - \frac{(n-1)(n-2)}{6n}\alpha^2.$$

Setting $\lambda = (\frac{\alpha}{4}n - \frac{\alpha}{2} - \frac{(n-1)(n-2)}{6n}\alpha^2)n^{-1/2} = \Omega(\alpha n^{1/2})$, we have $\mathbf{E}[Z] - \lambda n^{1/2} \geq r - n + \frac{\alpha}{4}n$, hence the lemma. □

Now we are ready to prove Lemma 1.

PROOF. (of Lemma 1) Consider a particular $f : [u] \to [r]$ and a random input $I_{\mathbf{u}}$. The probability that a randomly chosen key $x$ from $[u]$ has $f(x) = i$ is exactly $|f^{-1}(i)|/u$. This is exactly an $(n, r, \vec{\beta})$ bin-ball game where $\beta_i = |f^{-1}(i)|/u$. Let $Z$ be the number of empty bins at the end of such a bin-ball game. Note that we have $\eta_f(I) = n - (r - Z)$, which, by Lemma 1, does not exceed $\frac{\alpha}{4}n$ with probability at most $e^{-\Omega(\alpha^2 n)}$. Since there are $2^{m\log u}$ different $f$'s in $\mathcal{F}$, by a union bound, the probability that $I_{\mathbf{u}}$ is good for at least one $f \in \mathcal{F}$ is at most $e^{-\Omega(\alpha^2 n)} \cdot 2^{m\log u}$. Thus if $n > cm\log u/\alpha^2$ for some sufficiently large $c$, this probability is $e^{-\Omega(\alpha^2 n)} = o(1)$. □

## 4.3 Lower bound for the boundary-oblivious model

Now we prove the lower bound for the boundary-oblivious model, where the layout is required to work for any shifting $s$.

THEOREM 3. *For any fixed block size $b$, consider any hash table that stores $n$ uniformly random keys. There exists some shifting $s$ for which the hash table has an expected average successful search cost at least $1 + \frac{\alpha}{5b}$, for $n$ sufficiently large and $\alpha = \omega(n^{-1/2})$.*

PROOF. Consider any input $I \in \mathcal{I}$. Suppose that the hash table uses $f_I \in \mathcal{F}$ and $g_I$ on input $I$. Define $\gamma(s, I)$ to be the number of keys in $I$ that need two I/Os to search when the shifting is $s$, i.e., those keys $x$ with $f_I(x) \neq g_I(x)$ and $g'_I(x) = ib - s$ for some integer $i$. Note that the average search cost on $I$ is $1 + \gamma(s, I)/n$, and the expected average search cost on a random $I_{\mathbf{u}}$ is $1 + \mathbf{E}_{\mathbf{u}}[\gamma(s, I_{\mathbf{u}})]/n$, which we will show to be greater than $1 + \frac{\alpha}{5b}$.

Consider any $I \in \mathcal{I}_{\mathcal{F}}$. Since $I$ is bad for all $f \in \mathcal{F}$, it is also bad for $f_I$. Thus there are at least $\frac{\alpha}{4}n$ keys $x$ in $I$ with $f_I(x) \neq g_I(x)$. For these keys, $g'_I(x)$ is defined and there is exactly one $s$ such that $g'_I(x) = ib - s$ for some integer $i$. So we have $\sum_{s=0}^{b-1}\gamma(s, I) \geq \frac{\alpha}{4}n$. By Lemma 1, $I_{\mathbf{u}}$ belongs to $\mathcal{I}_{\mathcal{F}}$ with probability $1 - o(1)$, so

$$\sum_{s=0}^{b-1}\mathbf{E}_{\mathbf{u}}[\gamma(s, I_{\mathbf{u}})] = \mathbf{E}_{\mathbf{u}}\left[\sum_{s=0}^{b-1}\gamma(s, I_{\mathbf{u}})\right] \geq (1 - o(1))\frac{\alpha}{4}n \geq \frac{\alpha}{5}n.$$

By the pigeonhole principle, we must have one $s$ such that $\mathbf{E}_{\mathbf{u}}[\gamma(s, I_{\mathbf{u}})] \geq \frac{\alpha n}{5b}$, and the lemma is proved. □

## 4.4 Lower bound for the block-size-oblivious model

Next we give the lower bound under the block-size-oblivious model, in which the block boundaries are always multiples of $b$, but the layout of the hash table is required to work with any $b$. Since it is not possible to prove a lower bound of the form $1+\Omega(\alpha/b)$ for all $b$ (that would be a lower bound in the cache-aware model), instead we show that $1 + o(\alpha/b)$ is not achievable, i.e., the following is false: "$\forall\epsilon\exists n_0\exists b_0\forall n > n_0\forall b > b_0$, the cost is at most $1 + \epsilon\alpha/b$." In particular, we show that this statement is false for $\epsilon = \frac{1}{17}$.

THEOREM 4. *Consider any hash table that stores $n$ uniformly random keys. For any $b_0$, there exists a block size $b \geq b_0$ on which the expected average success search cost on $n$ keys is at least $1 + \frac{\alpha}{17b}$, for any $n$ sufficiently large and $\alpha = \omega((\log\log n)^{-1/2})$.*

We follow the same framework as in the proof of Theorem 3. Let $\rho(b, I)$ be the number of keys $x$ in $I$ with $f_I(x) \neq g_I(x)$ and $b|g'_I(x)$; these keys need two I/Os to search when the block size is $b$ in the block-size-oblivious model. On a random $I_u$, the expected average search cost is $1 + \mathbf{E}_{\mathbf{u}}[\rho(b, I_{\mathbf{u}})]/n$. From here suppose we were to continue to follow the proof of Theorem 3 and consider the summation of $\mathbf{E}_{\mathbf{u}}[\rho(b, I_{\mathbf{u}})]$ over all $b \in \{b_0, b_0 + 1, \ldots, r\}$. Each $x$ contributes 1 to the summation when $b = g'_I(x)$, so we still have $\sum_{b=b_0}^{r} \mathbf{E}_{\mathbf{u}}[\rho(b, I_{\mathbf{u}})] = \Omega(\alpha n)$. This, unfortunately, only guarantees the existence of a $b$ such that $\mathbf{E}_{\mathbf{u}}[\rho(b, I_{\mathbf{u}})]$ is at least $\Omega(\frac{\alpha n}{r})$ or $\Omega(\frac{\alpha n}{b\log r})$, where the latter uses the fact that $\sum_{b=b_0}^{r} 1/b = \Theta(\log r)$. Neither is strong enough to give us the desired lower bound. Below we show how we prove Theorem 4 by restricting $b$ to the primes and a much more careful analysis.

LEMMA 3. *Let $P_k$ be the set of all primes that are smaller than $k$, and let $P = P_r - P_{b_0}$ be the set of all primes that are in the range $[b_0, r)$. For $\alpha = \omega((\log\log n)^{-1/2})$, we have*

$$\mathbf{E}_{\mathbf{u}}\left[\sum_{b\in P} \rho(b, I_{\mathbf{u}})\right] = \sum_{b\in P} \mathbf{E}_{\mathbf{u}}[\rho(b, I_{\mathbf{u}})] > (1-o(1))\frac{\alpha}{16} n \log\log r,$$

*as $n \to \infty$.*

Note that Lemma 3 implies that there must be a $b \in P$ such that $\mathbf{E}[\rho(b, I_{\mathbf{u}})] \geq \frac{\alpha}{17b} n$, proving Theorem 4, since otherwise we would have

$$\sum_{b\in P} \mathbf{E}[\rho(b, I_{\mathbf{u}})] \leq \frac{\alpha}{17} n \sum_{b\in P} \frac{1}{b} \leq \frac{\alpha}{17} n(\log\log r + O(1)).$$

Here we use the following approximation for the prime harmonic series [22]:

$$\sum_{b\in P_r} \frac{1}{b} = \log\log r + O(1).$$

Thus $\sum_{b\in P} \mathbf{E}[\rho(b, I_{\mathbf{u}})] \leq \frac{\alpha}{17} n (\log\log r + O(1))$, contradicting Lemma 3.

### *Proof of Lemma 3.*

In the rest of this subsection we prove Lemma 3. We need the following fact from number theory. Let $\mu(s)$ denote the number of distinct prime factors of $s$.

LEMMA 4 ([22]). *Let $\xi(r) \to \infty$. Then*

$$\left|\left\{l \leq r : |\mu(l) - \log\log r| > \xi(r)\sqrt{\log\log r}\right\}\right| = O\left(\frac{r}{\xi^2(r)}\right).$$

PROOF. (of Lemma 3) By Lemma 1 we know that $I_{\mathbf{u}}$ belongs to $\mathcal{I}_{\mathcal{F}}$ with probability $1 - o(1)$, so it suffices to prove that for any $I \in \mathcal{I}_{\mathcal{F}}$,

$$\sum_{b\in P} \rho(b, I) > (1 - o(1))\frac{\alpha}{16} n \log\log r.$$

Consider any $I \in \mathcal{I}_{\mathcal{F}}$. Let $G$ be the set of distinct $g'_I(x)$'s for the keys $x \in I$. Let $\mu_P(s)$ be the number of distinct prime factors of $s$ that are in $P$. By definition $\mu_{P_{b_0}}(s)$ is the number of distinct prime factors of $s$ that are in $P_{b_0}$, and it follows that $\mu(s) = \mu_{P_{b_0}}(s) + \mu_P(s)$. Note that $\rho(b, I)$ is at least the number of multiples of $b$ in $G$, so we have

$$\sum_{b\in P} \rho(b, I) \geq \sum_{l\in G} \mu_P(l) = \sum_{l\in G} \mu(l) - \sum_{l\in G} \mu_{P_{b_0}}(l). \quad (2)$$

Next we show that $\sum_{l\in G} \mu(l)$ is large. Firstly, observe that

$$|G| > \frac{\alpha}{8} n. \quad (3)$$

This is because $I$ is bad for $f_I$, so at least $\frac{\alpha}{4}n$ keys in $I$ have $f_I(x) \neq g_I(x)$ and thus their $g'_I(x)$'s are defined. The $g_I(x)$'s for these keys must be distinct, and each $g'_I(x)$ is either $g_I(x)$ or $g_I(x) + 1$, so there are at least $\frac{\alpha}{8}n$ distinct $g'_I(x)$'s for the keys in $I$.

Secondly, by choosing $\xi(r) = \frac{(\log\log r)^{1/4}}{\sqrt{\alpha}}$ in Lemma 4 we get:

$$\left|\left\{l \leq r : \mu(l) \leq \left(1 - \frac{1}{\sqrt{\alpha}(\log\log r)^{1/4}}\right)\log\log r\right\}\right| = O\left(\frac{\alpha r}{\sqrt{\log\log r}}\right).$$

Since we require $\alpha = \omega(\frac{1}{\sqrt{\log\log n}})$ which implies $\frac{\alpha r}{\sqrt{\log\log r}} = \frac{\alpha n}{\alpha\sqrt{\log\log r}} = o(\frac{\alpha}{8}n)$ and $\frac{1}{\sqrt{\alpha}(\log\log r)^{1/4}} = o(1)$, it holds that for at least $|G| - o(1)\frac{\alpha}{8}n$ distinct $l \in G$,

$$\mu(l) > (1 - o(1))\log\log r. \quad (4)$$

By inequalities (3) and (4), we have

$$\sum_{l\in G} \mu(l) > (1 - o(1))\frac{\alpha}{8} n \log\log r. \quad (5)$$

It remains to upper bound $\sum_{l\in G} \mu_{P_{b_0}}(l)$. Note that for any $b \in P_{b_0}$, the number of integers in $[r]$ that are divisible by $b$ is at most $r/b$, so each $b$ will be counted at most $r/b$ times in $\sum_{l\in G} \mu_{P_{b_0}}(l)$. Hence,

$$\sum_{l\in G} \mu_{P_{b_0}}(l) \leq \sum_{b\in P_{b_0}} r/b = r\left(\log\log b_0 + O(1)\right).$$

Therefore, as long as $\alpha \geq \sqrt{\frac{32\log\log b_0}{\log\log n}} > \sqrt{\frac{16\log\log b_0}{\log\log n/\alpha}}$, we have

$$\log\log b_0 < \frac{\alpha^2}{16}\log\log\frac{n}{\alpha},$$

so

$$\sum_{l \in G} \mu_{P_{b_0}}(l) < \frac{\alpha}{16} n \log\log r + O(r)$$
$$= (1 + o(1)) \frac{\alpha}{16} n \log\log r. \quad (6)$$

Finally, combining (2), (6), and (5) completes the proof. □

### 4.5 Lower bounds on updates

Our lower bounds in this paper are concerned with the query cost only. How about updates? The blocked probing algorithm in Section 3 has an amortized update cost of $1 + O(1/b)$ I/Os, but can we improve it to $o(1)$ I/Os, possibly by buffering the updates in internal memory and write them to external memory in batches? A recent result by Wei et al. [24] has eliminated this possibility by proving a $1 - 1/2^{\Omega(b)}$ lower bound (in the cache-aware model) on the amortized update cost if the successful query cost is to be $t_q = 1 + 1/2^{\Omega(b)}$. Even more recently, Verbin and Zhang proved [23] that if $t_q$ is $o(\log_{b \log n} n)$ for both successful and unsuccessful queries, then the update cost has to be $\Omega(1)$. These results show that for external hashing, buffering is essentially useless and modifying the hash table on disk directly is the only way to perform updates.

## 5. OPEN PROBLEMS

An interesting open question is, although we have proved a matching lower bound in the cache-oblivious model, we do not yet know if $t_q = 1 + 1/2^{\Omega(b)}$ is optimal in the cache-aware model (or in the cache-oblivious model with the two more conditions). It is known that we can achieve $t_q = 1$ (namely, *perfect hashing*) with an internal memory of size $m = \Theta(n/b)$ [11, 16]. On the other hand, external linear probing and blocked probing achieve $t_q = 1 + 1/2^{\Omega(b)}$ with only $m = \Theta(b)$. There seems to be a tradeoff between $m$ and $t_q$ but this tradeoff is yet to be understood.

## 6. REFERENCES


[1] P. Afshani, C. Hamilton, and N. Zeh. Cache-oblivious range reporting with optimal queries requires superlinear space. In *Proc. Annual Symposium on Computational Geometry*, 2009.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The cost of cache-oblivious searching. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2003.

[4] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[5] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. ACM Symposium on Theory of Computing*, 2003.

[6] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[7] E. Demaine. Cache-oblivious algorithms and data structures. In *EEF Summer School on Massive Datasets*. Springer Verlag, 2002.

[8] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[9] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $o(1)$ worst -case access time. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 165–170, 1982.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[11] G. H. Gonnet and P.-Å. Larson. External hashing with limited internal storage. *Journal of the ACM*, 35(1):161–184, 1988.

[12] B. He and Q. Luo. Cache-oblivious databases: Limitations and opportunities. *ACM Transactions on Database Systems*, 33(2), article 8, 2008.

[13] M. S. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[14] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[15] P.-Å. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[16] P.-Å. Larson. Linear hashing with separators—a dynamic hashing scheme achieving one-access retrieval. *ACM Transactions on Database Systems*, 13(3):366–388, 1988.

[17] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proc. International Conference on Very Large Data Bases*, pages 212–223, 1980.

[18] M. Mitzenmacher and S. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2008.

[19] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] A. Pagh, R. Pagh, and M. Ružić. Linear probing with constant independence. In *Proc. ACM Symposium on Theory of Computing*, 2007.

[21] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[22] G. Tenenbaum. *Introduction to analytic and probabilistic number theory*. Cambridge Univ Press, 1995.

[23] E. Verbin and Q. Zhang. The limits of buffering: A tight lower bound for dynamic membership in the external memory model. In *Proc. ACM Symposium on Theory of Computing*, 2010.

[24] Z. Wei, K. Yi, and Q. Zhang. Dynamic external hashing: The limit of buffering. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2009.