Hashing Based Dictionaries in Different Memory Models

Zhewei Wei [*]

April 28, 2010

[*]Department of Computer Science and Engineering
The Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

Hashing based dictionaries are one of the most fundamental data structures in computer science, in both theory and practice. They have been intensively studied for decades, and numerous results have been obtained in many memory models. In this survey, we try to cover some basic approaches and techniques for designing hashing based dictionaries. We focus on three most popular memory models: the *RAM model*, the *I/O model* [1] and the *cache-oblivious model* [18]. Some lower bound results are discussed as well.

This survey is in partial fulfillment of the requirements for the Ph.D. qualifying examination in computer science, HKUST

Contents

1 Introduction

1.1 Problems

The *dictionary* is an important class of data structures in computer science. Given a subset S of a universe U, a dictionary stores S in a memory range R, such that the following queries can be answered efficiently:

- Membership: Given a *key* $x \in U$, does x belong to S?

A *static* dictionary is one that does not change over time. A *dynamic* dictionary should also support updates:

- Insertion: Include x into the dictionary;
- Deletion: Remove x from the dictionary.

In many applications each key of S is associated with some data, and insertion will include a pair (key, data) into the dictionary. The *lookup* operation is considered in these applications:

- Lookup: Does x belong to S? If so, what is its associated data?

Many methods were devised for dictionary problems. In this survey we study *hashing techniques*, which offer the highest performance for the basic dictionary operations. According to Knuth [24], the idea of hashing was originated 50 years ago by H.P. Luhn. The basic idea is to use a function $h : U \to R$, called *hash function*, to map the keys in S to a memory range R. It is possible that there are collisions, i.e., two different keys are mapped to the same memory location, and many collision resolution strategies were proposed to handle such situations. The key insight of hashing is, if the hash function is "good" enough, we do not expect many collisions and thus efficiency is achieved.

1.2 Models

To theoretically study the dictionary problem and the hashing techniques, we need a specific computational model that describes how computation takes place and what cost to be charged. In this survey we consider the three most popular models: the RAM model, the I/O model, and the cache-oblivious model. These models are used to describe *upper bounds*. If we aim at proving *lower bounds*, we turn to *the cell probe model* [54].

RAM Model The first model we consider in this survey is the *unit-cost word RAM*. In this model the memory is an array of bit strings called words. Each word consists of w bits, for a positive integer parameter w, called the *word length*. In the survey we assume the universe U is $\{0, \ldots, 2^w - 1\}$, or equivalently $w = \log |U|$. This assumption is sometimes referred to as the *trans-dichotomous* assumption [16]. For lookup operation, we assume the data has constant size so all results holds by considering a larger w. All computation takes place in CPU that has a constant number of words of registers, on which standard operations can be carried out on words in constant time. We adopt the *multiplication model*, whose instruction set includes addition, bit-wise boolean operations, shifts, and multiplication. We measure the space requirements of a word-RAM algorithm in units of w-bit words. Since accessing memory is much slower than executing instructions, in the hashing literature typically the algorithm's cost is measured only by the number of memory words it accesses.

Most hashing-based dictionaries require a source of random numbers. For randomized algorithms we equip the RAM model with a special instruction which assigns random and independent words to a register. In addition, many classical results of hashing-based dictionaries are obtained under the *uniform hashing model*, in which the algorithms have access to a truly random function that distributes the keys uniformly and independently.

I/O Model A major drawback of RAM is the lack of ability to capture the fundamental characteristics of modern hierarchical memory systems. As a consequence, a number of more elaborate models have been introduced in recent years. The I/O model, introduced by Aggarwal and Vitter [1] in 1988, is the most widely used one. In this model we consider a two-level memory hierarchy: a slow but conceptually unlimited *external memory* and a fast *internal memory* of size m. All computation takes place on data in internal memory, and the transfer of data between internal and external memory happens in blocks of b consecutive words; the complexity of an algorithm is the number of such *I/Os* (sometimes called *block transfers*) it performs. It is assumed that algorithms have complete control over transfers of blocks between the two levels. The I/O model captures the essence of the memory hierarchy when the memory transfer between two levels of the memory hierarchy dominates the running time, and it is sufficiently simple to make analysis of algorithms feasible. By now, a large number of results for the I/O model have been obtained — see the surveys by Arge [2] and Vitter [50].

Cache-Oblivious Model The main disadvantage of the I/O model is that the blocking must by programmed explicitly, resulting in programs that are less flexible to different-scale problems and that do not adapt well when the dominating memory level changes. 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 [18] due to its elegance and simplicity. The cache-oblivious model is very similar to the I/O model, the only difference being that a cache-oblivious algorithm is unaware of the parameter b and m. In other words, a cache-oblivious algorithm is formulated in the RAM model but analyzed in the I/O model, with the analysis required to hold for any b and m. The main idea of the cache-oblivious model is that by avoiding any memory-specific parametrization the cache-oblivious algorithm has an optimal number of memory transfers between all levels of an unknown, multilevel memory hierarchy. Thus the cache-oblivious model is effectively a way of modeling a complicated multi-level memory hierarchy using the simple two-level I/O-model.

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 [22].

Cell Probe Model The cell probe model, proposed by Yao [54], is usually used to show lower bounds. In this model, a data structure is a collection of b-bit cells, and the complexity of any operation on the data structure is just the number of cells that are read and/or changed. It is arguably the strongest computation model one can conceive for data structures; in particular it is at least as powerful as the RAM with any operation set. As a result, lower bounds obtained in this model are generally considered to be impassable. The cell size b is an important model parameter, and various b's have been considered, often leading to models with dramatically different characteristics. The case $b = 1$ (a.k.a. the *bit probe model*) yields a clean combinatorial model, but it is mainly for theoretical interests. The most studied case is $b = \log u$, where u is the universe size, is called *word probe model*, which corresponds to the trans-dichotomous assumption. When one considers a general (usually large) b, this corresponds to the I/O model.

In order to prove lower bound for hashing problems, two common assumptions are often presented: the first one is *indivisibility* assumption, i.e., the (key, data) pairs must be treated as atomic elements, they can only be moved or copied between cells in their entirety, and when answering a query, the query algorithm must visit the cell that actually contains the item or one of its copies. The next assumption is the presence of some *free cells*. For $b \leq \log u$ we usually assume these cells are used to store the hash function. For larger cell size it is possible that the algorithm uses these free cells to buffer insertions, in which case the free cells corresponds to the internal memory in the I/O model.

2 Preliminaries and Basic Techniques

2.1 Notations

Throughout this survey we use the following notations:

U	Universe, assumed to be $[u] = \{0, 1, \ldots, u-1\}$
S	Data set, which is a subset of U of size n
R	The hash table. In the RAM model and the cache-oblivious model R is assumed to be a consecutive memory with address $\{0, 1, \ldots, r-1\}$; In the I/O model, R is used to denote the block set $\{B_0, B_1, \ldots, B_{r-1}\}$.
m	The size of the internal memory (in words).
b	Block size. Without specification, we assume a block can accommodate b keys
α	Load Factor. In the RAM model and the cache-oblivious model, $\alpha = n/r$, and in the I/O model $\alpha = n/rb$.

2.2 Classical Results

Chaining and *linear probing* are the two oldest collision resolving techniques in the hashing world. For chaining, a collision is resolved by simply putting the collided elements in a linked list. Given a hash function h, a search for key x performs a linear search in the list $h(x)$. To insert x we simply append it to the list $h(x)$. In linear probing, a search for x successively probe positions $h(x), h(x)+1, \ldots, r-1, 0, 1, \ldots, h(x)-1$ until x is found or we encounter an empty position, in which case we know that x is not stored in the table. Insertion proceeds in the same manner, and we put x in the first empty position encountered. The analysis of chaining and linear probing, under the uniform hashing assumption, can be found in [24], and we summarize the results in Table 1.

	Successful Query	Unsuccessful Query
Chaining	$1 + \frac{\alpha}{2}$	$1 + \alpha$
Linear Probing	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$

Table 1. Expected cost for chaining and linear probing in the RAM model, up to an additive $O(1)$ term.

Hashing in the I/O model is often referred to as *external hashing*. Chaining and linear probing process in the same manner as in the RAM model, except that each bucket can store b keys. For the expected I/O cost of a lookup, chaining and linear probing share the same asymptotic formula in the I/O model:

$$C_n = 1 + O\left(e^{(1-\alpha+\ln\alpha)b}\right).$$

Note that the term in the big-Oh is exponentially decreasing with b as long as α is bounded from 1, indicating that the performance of external hashing benefits greatly from a large block size.

2.3 Hashing With Limited Independence

The uniform hashing assumption is unrealistic, since to simply store a truly random hash function requires $u \log r$ bits. To bridge the gap between hashing algorithms and their analysis Carter and Wegman introduced *universal hashing* [6]. The key idea is that, since a truly random function can be obtained by randomly selecting a function from the family of all functions that map U to R, it is possible to construct a much smaller function family $\mathcal{H}$ such that a function selected from $\mathcal{H}$ has similar performance guarantee. Our definition of universal hashing follows the convention in [45]:

Definition 1 *A family of hash functions $\mathcal{H}$ with domain U and range R is (k, c)-wise independent if it is finite and for all $y_1, y_2, \ldots, y_k \in R$, for all distinct $x_1, x_2, \ldots, x_k \in U$,*

$$|\{h \in \mathcal{H} : \, h(x_i) = y_i, i = 1, 2, \ldots, k\}| \leq c\frac{|\mathcal{H}|}{|R|^k}.$$

$(2, c)$-wise independent family is also referred to as *universal family*. It should be noted that this definition is slightly different from *k-wise independence*, which maps k keys to R independently. However this only leads to a constant difference in the hashing problems as long as c is a constant, so we will not distinguish k-wise and (k, c)-wise in the rest of the survey.

Carter and Wegman [52] exhibited the following families of $(k, (1+r/p)^k)$-wise independent hash functions where $U = [p]$, $R = [r]$, and p is a prime:

$$\mathcal{H}_k = \left\{h : \, h(x) = \Big((a_{k-1}x^{k-1} + \cdots + a_0) \bmod p\Big) \bmod r, \, a_j \in [p]\right\}.$$

This could be easily verified: observe that the family of degree $k - 1$ polynomials in the finite field Z_p is k-wise independent; to obtain a smaller range $R = [r]$ we may map integers in $[p]$ down to $[r]$ by a modulo r operation. This operation preserves independence, only making the family not uniform. However, as long as p is much larger than r the difference is only a small constant.

The above family presents very nice properties in many applications, however, the evaluation cost is high: to evaluate a single hash value requires $O(k)$ time. Many other families are proposed to solve this issue. In practice, the most popular pairwise family is Dietzfelbinger's multiply-shift scheme [9], which can be twice as fast as Carter and Wegman's family [48]. To hash w-bits integer to the range $r = 2^l$, Dietzfelbinger's family can be view as:

$$\mathcal{H} = \{h_a(x) : \, h_a(x) = (ax) >> (2w - l), a \in \{0, 1\}^{2w}\}$$

where $>>$ denotes unsigned shift. For higher independence, Siegel [45] showed that if U is not too large compared to k, there exists a $(k, 2)$-wise independent family that has *constant* evaluation time and uses space and initialization time $O(k^\epsilon)$, where ϵ is some constant. If one really wants the same independence as in the uniform hashing model, Pagh and Pagh [34] improved Siegel's family and show that a n-wise independent family, which performs exactly the same as truly random hash function on a n-key set, can be constructed in $O(n)$ time and space and evaluation takes constant time.

It is well known that 2-wise (or pairwise) independence guarantees expected $O(1)$ cost per operation for chaining. For linear probing, Siegel and Schmidt [42] showed that $\log n$-wise independence achieves the same performance for as full independence does. A recent breakthrough was by Pagh et al. [35], which showed that 5-wise independence suffices to obtain $O(1)$ cost per operation, thus giving the first practical implementation of linear probing with provable guarantees.

It might be worth noticing that there is another line to justify why hashing works well in practice: Mitzenmacher and Vadhan [32] showed that simple hash families can achieve the same performance as a truly random hash function if there is sufficient randomness in the data itself. Their method exploits the entropy in the data, and they show that a mild assumption on the entropy inside the data, which holds for many typical data streams, is sufficient to make the performance of a pairwise family comparable to that of a truly random hash function.

2.4 Balls-and-bins paradigm

The *balls-and-bins paradigm* is an abstraction that appears in many hashing papers. A balls-and-bins paradigm describes the process where n balls are placed in r bins. Classic analysis assumes that each ball goes to any bin independently and uniformly at random, corresponding to the truly random hash function assumption in the analysis of hashing. In this section we study some of the techniques in the balls-and-bins paradigm that help us get a better understanding of how hashing works.

2.4.1 Chernoff-Hoeffding Bounds

Chernoff-Hoeffding bounds (Chernoff [7] and Hoeffding [23]) are fundamental tools used in bounding the tail probabilities of the sums of bounded and independent random variables. In the balls-and-bins setting, n balls are placed into r bins independently and uniformly at random, and we are interested in the number of balls falling into a particular bin. Let X_i be the indicator for whether the i-th ball falls into our bin, and $X = \sum_{i=1}^{n} X_i$ is the number of balls in the bin after all balls have been placed. Since each ball goes to our bin with probability $1/r$, the expected number of balls in the bin is $E[X] = n/r$. Let $\mu = n/r$. By the Chernoff bound, the following tail bound holds:

$$Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}. \tag{1}$$

Chernoff-Hoeffding bound gives an exponential bound on the probability that an particular bin contains more than n/r balls, and thus is very useful in proving an expected cost that is very close to 1. However, the Chernoff-Hoeffding bound requires full independence of X_i, which means we can only use it under the uniform hashing assumption. The seminal paper of Schmidt et.al. [44] considered the problem of how much independence is needed to ensure the same bound. More specifically, it can be shown that if the X_i's are $\lceil \delta\mu \rceil$-wise independent, then the bound in (1) holds. They also give some very desirable bound for applications with even less independence. Theorem 1 presents two of their results:

Theorem 1 *Given n binary random variables $X_1, X_2, \ldots, X_n$, let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbf{E}[X]$. Then for any $\delta > 0$,*

1. *If the X_i's are $\lceil \mu\delta \rceil$-wise independent, then*

$$Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}. \tag{2}$$

2. *If the X_i's are k-wise independent for $k < \lceil \mu\delta \rceil$, then*

$$Pr[X > (1+\delta)\mu] < \left(\frac{kC}{e^{2/3}\delta^2\mu^2}\right)^{k/2}. \tag{3}$$

for $C \geq \max\{k, \mathbf{Var}(X)\}$ where $\mathbf{Var}(X)$ is the variance of X.

Note that $\mathbf{Var}(X)$ can be estimated in our balls-and-bins model: first observe that $\mathbf{Var}(X) = \sum_{i=1}^{n} \mathbf{Var}(X_i)$ if $k \geq 2$. Assuming uniformity, $\mathbf{Var}(X_i) = \mathbf{E}[(X_i - 1/r)^2] = 1/r(1 - 1/r) \leq 1/r$ and thus $\mathbf{Var}[X] \leq n/r = \mu$. So the bound in (3) holds for $C \geq \max\{k, \mu\}$.

2.4.2 Occupancy Problem

In a $(n, r, \vec{\beta})$ *occupancy problem*, we throw n balls into r bins independently at random. The probability that a ball goes to the j-th bin is β_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 1 *In an $(n, r, \vec{\beta})$ occupancy problem, $\Pr[Z \leq r - n + \frac{\alpha}{4}n] \leq e^{-\Omega(\alpha^2 n)}$, where $\alpha = n/r$.*

PROOF: Note that if $\vec{\beta}$ is the uniform distribution, the lemma can be proved using properties of martingales [33]. 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

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

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. ■

Lemma 1 can be used to prove lower bounds for hashing problems under different models, as we shall see in Section 6.

2.4.3 Poisson Approximation

A very powerful tool to analyze the uniform hashing model is the so called *Poisson approximation*. The key insight is that if we distribute n balls into r bins independently and uniformly at random, the number of keys received by a particular bin is a random variable that follows binomial distribution. A binomial distribution can be approximated by a Poisson distribution when n is very large. Below is the definition of Poisson approximation:

Definition 2 *The number of balls X falling in a bin with bucket size b can be approximated by a Poisson distribution:*

$$\Pr[X = k] = e^{-\alpha b}\frac{(\alpha b)^k}{k!}, \quad k = 0, 1, \ldots$$

where $\alpha = n/rb$.

The analysis for hashing is much simpler in the Poisson approximation. For example, the expected search cost for external memory chaining can be estimated by

$$\begin{aligned} C_n &= 1 + \sum_{k=b+1}^{\infty} (k-b)\Pr[X=k] \\ &= 1 + e^{-\alpha b}\frac{(\alpha b)^{b+1}}{(b+1)!}\left(1 + 2\frac{\alpha b}{b+2} + 3\frac{(\alpha b)^2}{(b+2)(b+3)} + \dots\right) \\ &\leq 1 + \frac{1}{(1-\alpha)^2}\cdot e^{(1-\alpha+\ln\alpha)b}. \end{aligned}$$

A mathematical transform, developed by Gonnet and Munro [20], builds the bridge between Poisson approximation and the exact balls-and-bins paradigm. Their work essentially says that any expected value $f(\alpha)$ obtained in the Poisson model can be transformed to the exact binomial model, by using $n^{\underline{i}}/(rb)^i$ to substitute α^i in the Maclaurin expansion of $f(\alpha)$, where $n^{\underline{i}} = n(n-1)\cdots(n-i+1)$.

3 RAM Model

3.1 Linear Probing with Constant Independence

The analysis of hashing with linear probing is usually considered as the birth of analysis of algorithms [24], and it is still attracting a lot of attention nowadays. However, Knuth's analysis relies on the uniform hashing assumption. The first known result on hashing with linear probing was given by Siegel and Schmidt in [43], where they show that $O(\log n)$-wise independence is sufficient to achieve essentially the same performance as the truly random function does. The main reason for this is that, by Theorem 1, $O(\log n)$-wise independent events has probability bounds only differing from the full independence case by $O(1/n)$.

However, $O(\log n)$-wise hash function is still expensive to evaluate. Recently a breakthrough result, published by Pagh et al [36], showed that constant independence is sufficient to achieve constant bound. More precisely, they showed that 5-*wise independence* is enough to ensure a constant expected cost per operation, while pairwise independence may lead to *logarithmic* cost per operation. This result indicates simple and efficient hash functions with description stored in CPU registers can be used to give provable good expected performance.

Below we show that 5-wise independent family is sufficient to guarantee constant expected cost per operation. Our proof follows the one in [40] for the sake of simplicity. We will need the 4th moment inequality here:

Lemma 2 *Suppose we distribute n balls into r bins uniformly, X_i is a random variable indicating that the i-th ball falls into the first bin and $X = X_1 + \ldots + X_n$ is a random variable indicating the number of balls that fall into the first bin. Then $\mu = \mathbf{E}[X] = n/r$. If the X_i's are 4-wise independent, we have*

$$Pr[X \geq 2\mu] = O(1/\mu^2)$$

Note that this lemma is a simple corollary of Theorem 1. To apply moments to linear probing, we consider a perfect binary tree spanning the array. For notational convenience, let us assume that the load factor is at most $1/3$, i.e. $n \leq r/3$. A node on level l has 2^l array positions under it, and we expect $2^l/3$ keys to be hashed to one of them (but more or less keys may actually appear in the subtree, since items are not always placed at their hash position). Call the node *dangerous* if at least $\frac{2}{3}2^l$ keys hash to it. In the first stage, we will bound the total time it takes to construct the hash table (the cost of inserting n distinct keys). If the table consists of runs of $k_1, k_2, \ldots$ elements ($\sum k_i = n$), the cost of constructing it is bounded from above by $O(k_1^2 + k_2^2 + \ldots)$. To bound these runs, we make the following crucial observation: if a run contains between 2^l and 2^{l+1} elements, then some node at level $l-2$ above is dangerous.

For a proof, assume the run goes from positions i to j. The interval $[i, j]$ is spanned by 4 to 9 nodes on level $l-2$. Assume for contradiction that none is dangerous. The first node, which is not completely contained in the interval, contributes less than $\frac{2}{3}2^l$ elements to the run (in the most extreme case, this many elements are hashed to the last location of that node). But the subsequent nodes all have more than $32^{l-2}/3$ free locations in their subtree, so 2 more nodes absorb all excess elements. Thus, the run cannot go on for 4 nodes, a contradiction.

This observation gives an upper bound on the cost: add $O(2^{2l})$ for each dangerous node at some level l. Denoting by $p(l)$ the probability that a node on level l is dangerous, the expected cost is thus $\sum_l (n/2^l) \times p(l) \cdot 2^{2l} = \sum_l n \times 2^l p(l)$. Using the 2nd moment to bound $p(l)$, one would obtain $p(l) = O(2^{-l})$, so the total cost would be $O(n \lg n)$. However, the 4th moment gives $p(l) = O(2^{-2l})$, so the cost at level l is now $O(n/2^l)$. In other words, the series starts to decay geometrically and is bounded by $O(n)$. To bound the running time of one particular operation (query or insert x), we actually use the stronger guarantee of 5-independence. If the query lands at a uniform place conditioned on everything else in the table, then at each level it will pay the "average cost" of $O(1/2^l)$, which sums up to $O(1)$.

3.2 Perfect Hashing

Chaining with universal hashing and linear probing with 5-wise independence have a solid theoretical basis. However, these bounds are still *expected* constant, and bad performance could very well occur. *Perfect hashing* is a technique that achieves worst-case performance guarantee, and in this section we study two most famous perfect hashing constructions.

3.2.1 FSK Scheme

Fredman, Komlós and Szemeredi [17] published the first construction of a perfect hashing that uses linear space and processes the lookup operation using $O(1)$ time in the worst case. This is the first result that shows static *efficient* dictionary is indeed possible and this data structure is often referred to as the *FKS scheme*.

Suppose we want to store a set $S \in U$ of size n in an array of size r. Assume for simplicity $p = u$ is a prime. The family used is Carter and Wegman's universal family:

$$\mathcal{H}_r = \{h_k : U \to R |\, h_k(x) = (kx \bmod p) \bmod r\}.$$

Definition 3 *Consider any $S \in U$ with $|S| = n$, and let $R = \{0, \ldots, r-1\}$ with $r \geq n$. For each $i \in R$, the number of elements $x \in S$ colliding at i is denoted as*

$$B_i(k, r, S) = \{x \in S \mid \; h_k(x) = i\}$$

and its size is denoted by $b_i(k, r, S) = |B_i(k, r, S)|$.

Note that since $k \in U$, the description of a hash function h_k can be encoded into a key value in $U = \{0, 1, \ldots, u-1\}$ and stored in a single cell. The main lemma used by Fredman et al. is that if a function h_k is chosen from $\mathcal{H}_s$ uniformly at random, then

$$\mathbf{E}\left[\sum_{i=0}^{r-1} \binom{b_i(k, r, S)}{2}\right] \leq \frac{n(n-1)}{r}, \tag{4}$$

and by Markov's inequality

$$\Pr\left[\sum_{i=0}^{r-1}\binom{b_i(k,r,S)}{2} < \frac{2n(n-1)}{r}\right] \le \frac{1}{2}. \tag{5}$$

There are two special cases of this result. In the sparse case we choose $r = 2(n-1)$, relation (4) implies that for at least half of the function $h \in \mathcal{H}_r$ one has

$$\sum_{i=0}^{r-1}\binom{b_i(k,r,S)}{2} < n.$$

Such a function is used as a primary hash function which partitions S into buckets $B_i(k,r,S)$, $0 \le i \le r-1$. For each bucket $B_i(k,r,S)$ one store the elements of $B_i(k,r,S)$ in a secondary hash table of size $r_i = \max\{1, 2b_i(k,r,S)\,(b_i(k,r,S)-1)\}$. A secondary hash function h_{k_i} is chosen from $\mathcal{H}_{r_i}$ to guarantee no collision in the secondary hash table, where

$$\mathcal{H}_{r_i} = \{h_{k_i} \,|\, h_{k_i}(x) = (k_i x \bmod p) \bmod r_i\}.$$

Using relation (4) it follows that for at least half of the functions $h_{k_i} \in \mathcal{H}_{r_i}$ one has

$$\sum_{j=0}^{r_i-1}\binom{b_j(k_i, r_i, B_i(k,r,S))}{2} < 1,$$

implying that $b_j(k_i, r_i, B_i(k,r,S)) \le 1$ for all j. Therefore at least half of the functions in $\mathcal{H}_{r_i}$ are perfect on $B_i(k,r,S)$. The query of an element x is processed in the following manner: one first computes $h_k(x)$ which leads to the memory cell that stores the description of the secondary hash function h_{k_i}, and $h_{k_i}(x)$ is further computed to determine where x resides in the secondary hash table. The total space requirement is linear since

$$\sum_{i=0}^{r-1} r_i \le r + 4\sum_{i=0}^{r-1}\binom{b_i(k,r,S)}{2} = O(n).$$

Theorem 2 *There is a static hash table in the RAM model that stores n keys using $O(n)$ space and supports lookup operation in worst case $O(1)$ cost.*

Deterministic Construction of the FSK Structure It is not hard to see that FSK structure can be constructed in expected $O(n)$ time. If one wants to construct the structure deterministically, the problem becomes harder. The best known result to date is is by Hagerup et al. [21], who gave an $O(n \log n)$ time algorithm with linear space. This result is interesting because it can be translated to a dynamic dictionary, supporting insertions and deletions, by standard *logarithmic method* [46]. Particularly, the resulting dynamic dictionary supports lookups in time $O(\log\log n)$ and insertion $n^{1/\log\log n}$, which is a new lookup-insertion combination for deterministic linear space dictionaries.

3.2.2 Dynamic Perfect Hashing

The FKS scheme can be made dynamic, supporting insertions and deletions in amortized expected constant time [10]. The main techniques used are *partial rebuilding* and *global rebuilding*.

More specifically, to hash a n-key set S, we follow the FSK scheme and make the following relaxations: the size of the primary hash table can change between n and $2n$, and the size of a secondary hash table B_i can change from r_i to $2r_i$. Searching in this data structure is processed in the same way as for the FSK scheme, which gives the $O(1)$ worst-case guarantee. Deletion is processed using a technique called *deleting signal*, which only marks the deleting key and defer the actually deletion to the next global rebuilding. The most complicated part of the data structure is the insertion algorithm. In most cases the insertion is processed as a lookup; if the querying memory cell is empty, the key is inserted there. However, several things can go wrong during the insertion:

1. If the number of keys n exceeds the size of the primary hash table, we double the size of the primary hash table, and perform a global rebuilding.
2. If the total size of the hash table becomes too large (say, cn for some constant c), a global rebuilding is performed.
3. If the size of a secondary hash table B_i needed exceeds the current size r_i, we double the size of B_i, and perform a partial rebuilding on B_i.
4. If a collision happens in a secondary hash table B_i, a partial rebuilding is performed on B_i.

It can be shown that the amortized expected cost for an insertion is $O(1)$. The formal analysis is too technical and complicated so we skip it here, but the key observation is that, by doubling the prime hash table and the secondary hash tables, the probability that any of the above cases happens is very small and thus rehashing will only add an amortized expected $O(1)$ cost.

3.3 Cuckoo Hashing

The dynamic perfect hash table described in the last section has very desirable performance guarantee, but it is too complicated for implementation. *Cuckoo hashing*, developed by Pagh and Rodler [39], possesses the same theoretical properties as Dietzelbinger's dictionary, and is much simpler.

Theorem 3 *Cuckoo hashing stores a n-key subset of a universe U and supports lookup operations using worst-case 2 independent memory accesses, which can be done in parallel. Updates are processed in amortized expected constant time, and the space usage is $2(1+\epsilon)n$ words, where ϵ is a constant.*

We start with a static version of cuckoo hashing, proposed by Pagh [38]. The dictionary uses two hash tables, T_1 and T_2, each of length r, and two hash functions h_1, $h_2 : U \to R = \{0, \dots, r-1\}$. Each key $x \in S$ is stored in cell $h_1(x)$ of T_1 or $h_2(x)$ of T_2, but never in both. The lookup of a key x is simply probing both $h_1(x)$ and $h_2(x)$. It can be shown that if $r \geq (1+\epsilon)n$ for some constant $\epsilon > 0$, and h_1, h_2 are picked uniformly at random from an $(O(\log n), O(1))$-wise independent hash family, the probability that there is no way of arranging the keys of S according to h_1 and h_2 is $O(1/n)$. A suitable arrangement of the keys was shown in [38] to be computable in expected linear time, by a reduction to 2-SAT.

We now consider the dynamization of the above scheme. Deletion is of course simple to perform in constant time, not counting the possible cost of shrinking the tables if they becomes too sparse. Insertion is processed in the "cuckoo" manner, i.e. we first insert x to T_1 by looking at $h_1(x)$. If $h_1(x)$ is empty we put x there. Otherwise, suppose y is stored in $h_1(x)$. The algorithm evicts y, stores x in $h_1(x)$, and insert y to T_2 in the same way. It may happen that this process loops, and a value $MaxLoop$ set to $\Theta(\log n)$ is used to bound the number of iterations. When the number of keys evicted hits $MaxLoop$, we simply pick h_1 and h_2 again and do a rehashing.

In the rest of the section we analyze the running time of an insertion. The key observation is that when inserting a new element, we never examine more than $O(\log n)$ keys. Since our functions are $O(\log n)$-

independent, we can treat them as in the uniform hashing model. Also, to make the calculation easy, we assume that $r = 4n$.

Consider the process of inserting a key x_1. We will consider the probability that the insertion evicts at least $2t$ keys $x_1, y_1, \ldots, x_t, y_t, \ldots$. The behaviors of an insertion in the cuckoo hash table can be represented by the cuckoo graph G, whose vertex set is T_1 and T_2 and whose edge set contains edges of the form $(h_1(x), h_2(x))$ for all $x \in U$. In this way, the process of inserting item x_1 can be viewed as a walk on G starting at $h(x_1)$. There are three possible situations:

- **No cycle:** The simplest case is that the probing sequence finds an empty cell and no cycle happens(see Figure 1).

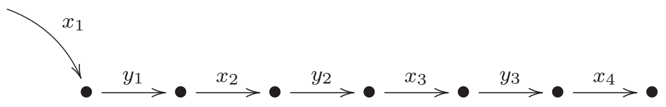

Figure 1. No cycle

  The first eviction happens if and only if $T[h_1(x_1)]$ is occupied. By the union bound it happens with probability at most

$$\sum_{x \in S, x \neq x_1} (\Pr[h_1(x) = h_1(x_1)] + \Pr[h_2(x) = h_1(x_1)]) < 2\frac{n}{4n} = \frac{1}{2}.$$

  Similarly, the next eviction would happen with the same probability. Thus the probability that two evictions happens is at most 2^{-2}. And we can argue that the process carries out the at least $2t$ evictions with probability at most 2^{-2t}.

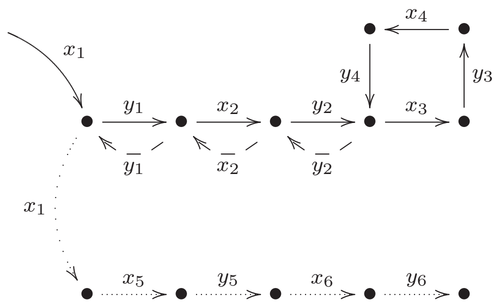

Figure 2. One cycle

- **One cycle:** It is possible that the probing sequence comes back to a cell that has already been visited. For example, if $h(y_j)$ is occupied by a previously evicted key y_i when we first attempt to insert y_j, the keys $y_i, x_i, \ldots, x_1$ will be evicted in that order, and x_1 will be sent to $T[h_2(x)]$, and the sequence continues. If it hits an empty position, the probing sequence would have exactly one cycle, see Figure 2.

  In order to bound the probability of this probing sequence, we claim that in $x_1, y_1, \ldots, x_t, y_t$ there exists a consecutive subsequence distinct items of length at least $2t/3$ that starts or ends with x_1. The reason should be obvious; the sequence can be partition into 3 parts that start or end with x_1, and one of them must contains at least $2t/3$ keys. By the same reasoning as in the previous case, we have that the probability that the insertion process evicts all these keys is at most $2^{-2t/3}$.

- **Two cycles:** If a probing sequence with one cycle hits a previously visited cell again, another cycle is formed. In this case the process continues indefinitely if we do not require that it stops after $MaxLoop$ evictions, see figure 3.

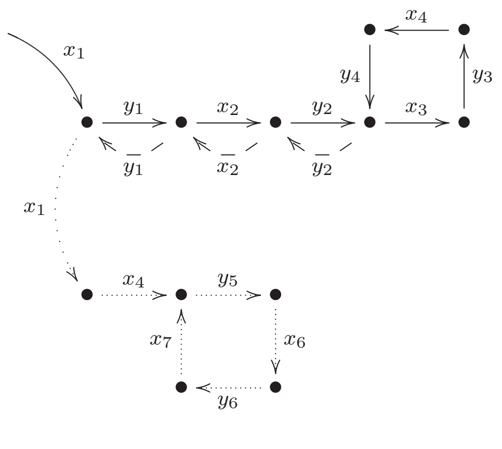

Figure 3. Two cycles

The probability of a sequence of length $2t$ with two cycles can be bounded by a union bound on all possible two-cycle configurations.

  - The first item in the sequence is x_1.
  - We have at most $(n-1)(n-2)\cdots(n-2t+1) < n^{2t-1}$ choices of other items in the sequence.
  - We have at most $2t$ choices for when the first loop occurs, at most $2t$ choices for where this loop returns on the path so far, and at most $2t$ choices for when the second loop occurs.
  - We also have to pick $2t-1$ hash values to associate with the items. (The sequence has $2t$ edges, but only $2t-1$ vertices. See Figure 3.)

Thus, there are at most $8t^3 n^{2t-1}(4n)^{2t-1}$ configurations. We have to choose the value of h_1 and the value of h_2 of each key, so the probability that a configuration occurs is given by $2^{2t}(4n)^{-4t}$. The 2^{2t} factor is due to the fact that edge (u,v) can correspond to either $h_1(x)=u, h_2(x)=v$ or $h_1(x)=v, h_2(x)=u$. Thus, the probability that a two-cycle configuration occurs is at most

$$\frac{8t^3 n^{2t-1}(4n)^{2t-1}2^{2t}}{(4n)^{4t}} = \frac{t^3}{n^2 2^{2t-1}}.$$

So for no cycle or one cycle case, the expect cost for a insertion can be bounded by $\sum_{t=0}^{\infty} 2^{-2t/3}(2t+1) = O(1)$. If the two cycle case happens the insertion will evicts $MaxLoop$ keys and rehash. Therefore, the probability that a two-cycle occurs at all is at most

$$\sum_{t=1}^{\infty} \frac{t^3}{n^2 2^{2t-1}} = O\left(\frac{1}{n^2}\right).$$

Also, the insertion process might stop and initiate rehashing after evicting $O(\log n)$ items while being in the no-cycle or the one-cycle configuration. In this case, we can set $MaxLoop$ to be $6\lg n$, so that the probability of this happening it at most $2^{-(6\lg n)/3} = 2^{-2\lg n} = n^{-2}$.

So, an insertion causes the data structure to rehash with probability $O(1/n^2)$. Therefore, n insertions can cause the data structure to rehash with probability at most $O(1/n)$. Thus, rehashing, which is basically n

insertions, succeeds with probability $1-O(1/n)$, which means that it succeeds after a constant number trials in expectation. In a successful trial, every insertion must fall into the first two cases. Therefore, a successful trial takes $n \times O(1) = O(n)$ time in expectation. In an unsuccessful trial, however, the last insertion can take $O(\log n)$ time, so it takes $O(n) + O(\log n) = O(n)$ time in expectation as well. Since we are bound to be successful after a constant number of trials, the whole process of rehashing takes $O(n)$ time in expectation. Hence, the expected running time of an insertion is $O(1) + O(1/n^2) \cdot O(n) = O(1) + O(1/n) = O(1)$.

Cuckoo Hashing and Randomness A big open question evolving cuckoo hashing is that, how much randomness do we need to make cuckoo hashing work? Until now there is a huge gap between the upper bound and the lower bound: A recent advance by Cohen and Kane [8] demonstrates that the 5-independence (which is slightly different than but close to 5-wise independence) is insufficient for constant amortized cost per operation for cuckoo hashing with 2 choices, but also show that only one of the two hash functions needs to be $\log n$-wise independent to obtain constant expected time per operation. Interestingly, they show that $\Theta(\log n)$-*joint-independence* are needed for cuckoo hashing to work within the same time bounds, where k-joint-independence are defined as follow:

Definition 4 *A family of pairs of hash functions h_1, h_2 is k-joint-independent if for any k distinct keys $x_1, x_2, \ldots, x_k$ and $2k$ values $a_1, a_2, \ldots, a_k$ and $b_1, b_2, \ldots, b_k$, for a randomly chosen pair of hash functions,*

$$\Pr[\forall i : \; h_1(x_i) = a_i \text{ and } h_2(x_i) = b_i] = \frac{O(1)}{r^{2k}}.$$

4 I/O Model

Hash tables are commonly used for indexing in large database management systems. In this case the hash tables are stored on external memory and the goal is to minimize the number of I/Os to retrieve or insert (delete) an element. The model considered here is the classical I/O model of Aggarwal and Vitter [1]. The hash table works especially well in the I/O model, since collisions happen only when there are more than b elements hashed to the same block. Large blocks help to push the performance of external hash tables to the limit: Using some common collision resolution strategies such as linear probing or chaining, the expected cost of a query is merely $1 + 1/2^{\Omega(b)}$ I/Os, provided the load factor α is a constant bounded from 1, as showed before. Most external hashing results assume the uniform hashing model, which we will adopt in this section.

4.1 Dynamic External Hashing

A challenge for designing external hash table is that the data set S can change drastically. For example, a hash index of a table can expand or shrink with the table. In this case maintaining the hash table size proportional to the size of S becomes crucial. Of course the standard doubling/halving strategy can be used to maintain the load factor α in the range $1/2 - \epsilon/2 \leq \alpha \leq 1 - \epsilon$ as we insert and delete keys in the hash table where $\epsilon > 0$ is any small constant. However if we want to improve the space utilization by maintaining a higher load factor, or if we want the hash table expand and shrink more smoothly with S, better strategies are needed. Various external dynamic hashing schemes have been proposed [28] for these purposes. Among these, *linear hashing* [29] and *extendible hashing* [13] appear to be the most commonly used.

Linear Hashing Linear hashing is a technique of maintaining high factor for external hash tables with chaining. The name linear hashing is used because the number of buckets grows or shrinks in a linear fashion.

For simplicity, we assume $h_0(x) = x \bmod r$ is the initial hash function. Insertions and deletions are performed using h_0 until a bucket overflow happens. When the first overflow occurs, the first block, B_0, is split into two blocks B_0 and B_r using a new hash function $h_1(x) = x \bmod 2r$, where B_r is a new block added to the hash table. A new overflow of some block triggers the split of B_2 and so on so forth, until B_{r-1} is split which is a sign of *full expansion*, and we start with B_0 to B_{2r-1} in the next round. If a pointer p is used to indicate which block is next to be split, we can easily identify the hash function for searching.

One major drawback of linear hashing is that a block may wait for a long time before getting split. This will create a very uneven distribution of the load over the hash table, causing a large number of overflow keys. The development of *Linear hashing with partial expansion* [26] is motivated to overcome this drawback. The blocks of the hash table are partitioned into groups, such that the size of any two groups differs by at most 1. The hash function values are distributed uniformly on the groups (i.e., the probability to have a hash function value in a particular group is the same for all groups), and are also distributed uniformly on the blocks within a group. To increase the number of blocks in the hash table by 1, the number of blocks in one of the groups is increased by 1, and the corresponding buckets are reorganized by hashing to the larger range. Decreasing the number of blocks in the hash table by 1 is done analogously. At the time where all groups have the same size, we may split all groups into two groups, or merge pairs of groups in order to maintain a group size of $\Theta(n_0)$, for some parameter n_0. The following theorem by Larson [27] describes the performance of linear hashing with partial expansion:

Theorem 4 *Linear hashing with partial expansions keeps the expected load on all buckets within a factor $1 + O(1/n_0)$ of each other, while adding an amortized expected cost of n_0/b I/Os per update cost, in which n_0 is the group size maintained.*

Extendible Hashing *Extendible hashing*, developed by Fagin et al. [13], is widely use in the database system due to its simplicity of expansion. Extendible hashing achieves worst case 1 I/O lookup cost, while maintaining a load factor of 69% in expectation.

Roughly speaking, extendible hashing stores an *directory* in the internal memory to direct the query.The directory has 2^d entries; each entry is a pointer to an external memory block. Let h be a truly random hash function defined on U, extendible hashing takes the d least significant bits of $h(x)$ as the index of x to the directory. The parameter d is called the *global depth* of the directory, and is chosen to be the smallest number such that no more than b keys are mapped to the same entry. This property is satisfied with high probability if the number of bits in $h(x)$ exceeds $3 \log n$.

Note that there does not have to be a distinct block for each of the 2^d directory entries. Several entries that share the same d' least significant bits can point to the same block if no more than b keys are hashed to these entries. And d' is the *local depth* of this block that specifies the number of bits used to identify the content in the block. When a block of local depth d' overflows it is split into two blocks with local depth $d' + 1$. In case $d' = d$ we need to double the directory. Conversely, if two blocks with the same local depth d' and contains the keys with the same d' least significant bits by applying h, these blocks are merged. If all blocks have local depth smaller than the global depth, the size of the directory is halved.

Clearly extendible hashing provides lookups in worst-case 1 I/O. The expected load factor α is $1/\ln 2 \approx 0.69$, which means that about 69% of the space is utilized. Flajolet [14] showed that the expected number of entries in the directory is $O(\frac{n^{1+1/b}}{b})$, meaning extendible hashing uses a internal memory of size superlinear to the number of blocks in the hash table.

4.2 External Perfect Hashing

External perfect hashing, i.e., data structures that perform lookups using worst-case 1 I/O, is very desirable in the database community, due to the high cost of disk accesses. Many strategies were proposed to achieve this goal (for example, extendible hashing is perfect), and below we summarize some important results.

Signature Hashing *Signature hashing*, proposed by Gonnet and Larson [19], achieves worst-case 1 I/O lookup using an internal memory size proportional to n/b. The technique can be used on different collision resolving strategies. Take chaining for example. When a block overflows, a new block is chained to the bucket. In order to perform lookup in 1 I/O we need to further separate the blocks in this linked list. In signature hashing, some *hashing signatures* are stored in the internal memory for this purpose. The hashing signature is usually obtained by using a secondary hash function to hash the keys in the same bucket to $[0, 1)$ and taking the first k bits. For each overflowed block, the largest signature of this block is stored in the internal memory, and is referred to as the *separator* of this block. A search for key x will find the corresponding bucket, find the block with the smallest separator that are larger than x's signature, and search that block.

The main problem with signature hashing is that the internal memory size is too large. In the worst case it could be n/b bits. Employing the uniform hashing assumption, the internal memory size can be reduced to $n/2^{\Omega(b)}$ words in expectation.

B-perfect Hashing Mairson [30] considered implementing a b-perfect hash function, which ensures that no more than b keys are hashed to the same block. The structure is also minimal, meaning the number of blocks is only $\lceil n/b \rceil$. Mairson showed that such a function can be implemented using $n \log b/b$ bits of internal memory. A matching lower bound was showed if the minimal condition is required. Mairson's technique, however, is based a probabilistic argument and therefore highly nonconstructive.

4.3 Lookup Using 2 Parallel I/Os

Multi-choice hashing scenario is found to be particularly useful in the I/O model, since one can easily perform the searches in parallel using multiple disks. One hashing scheme called *two-way chaining* was introduced by Azar et al. [5]. It can be thought of as two chaining hash tables with two independent hash functions h_1 and h_2. A search for key x will go through all the keys residing in $h_1(x)$ and $h_2(x)$ until x is found. New keys are always inserted to the bucket with less keys. It can be shown that the the expected overflow size is $n/2^{2^{\Omega(1-\alpha)b}}$, which is doubly exponentially in the average number of free spaces in each block.

Cuckoo hashing is also analyzed in the I/O model [11, 15], where the large block size helps to improve the space utilization. More specifically, in [11] Dietzfelbinger and Wiedling showed that if each block can accommodate b keys, then the load factor of cuckoo hashing can be improved from $1/2$ to $1 - 1/2^b$.

5 Cache-Oblivious Model

Pagh et al [41]. considered how to build a cache-oblivious hash table such that the $1 + 1/2^{\Omega(b)}$ bound can be achieved. A straightforward way of making the hash table cache-oblivious is to simply use linear probing but ignoring the blocking at all.[2] The I/O cost of linear probing is analyzed, which turns out to delicately depend on C_n and C'_n, the expected number of probes in a successful and unsuccessful search, respectively.

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

Theorem 5 *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*

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

Next Pagh et al. show that the *blocked probing* algorithm[35] 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. A lower bound is presented to show that neither condition is dispensable. Specifically, they 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.

5.1 Blocked Probing

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 [35] the *distance measure* $d(x, i_x)$ is defined 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 $I(x, j) = \{i \mid d(x, i) \leq j\}$. 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 $|I(x, j)| = 2^j$.

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.

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., $X_j = |\{y \in S \mid h(y) \in I(x, j)\}|$. Note that the algorithm will not visit any locations outside of $I(x, j^*)$, where $j^* = \min\{j \mid X_j \leq 2^j\}$. 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 $\Pr[X_j - 1 > 2^j - 1] \leq 2^{-(1-\alpha)^2 (2^j - 1)/2}$, 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 α bounded away from 1.

5.2 Cache-oblivious dynamic hash tables

The resizing technique from the previous sections can be made cache-oblivious to maintain the load factor of $\alpha = 1 - \Theta(\varepsilon)$. Suppose initially r is a power of 2 and $n > (1 - 2\varepsilon)r$. Adjust ε so that εr is also a power of 2; this will not change ε 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 εr 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. When r doubles or halves, we rebuild the entire hash table using a new part size εr. 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.

The following theorem summarizes their main results:

Theorem 6 *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 $\alpha \geq 1 - \varepsilon$ for any constant $\varepsilon > 0$.*

6 Lower Bound

6.1 Cell Probe Complexity of Perfect Hashing

Mehlhorn [31] showed that minimal perfect hashing requires space $\Omega(n)$ bits. The lower bound holds even if the range of the functions r is of size $O(n)$ rather than exactly n, and h is required to be injective. Below we demonstrate a simple proof of this result using Lemma 1:

First note that this lower bound is actually on the size of any *perfect hash family*. More precisely, a perfect hash family $\mathcal{H}$ consists of some function $h : U \to R$, such that given any n-subset S of U, we can find a h in $\mathcal{H}$ to map S to R without collision. Mehlhorn's result says that the size of $\mathcal{H}$ is at least $\Omega(2^n)$. Consider a fixed mapping $f : U \to R$ and a random subset S, the probability of f being perfect for S is exact the probability that f leaves $r - n$ cells empty after S is mapped. By Lemma 1 we know that this probability is $e^{-\Omega(n)}$, and by union bound we need at least $e^{\Omega(n)}$ functions to map all possible S.

Mairson [30] considered a related generalization of minimal perfect hashing which can be translated to a lower bound on the internal memory size of any external perfect hashing algorithm. He proved that if each block can accommodate b keys, and the total number of blocks is minimal: $\lceil n/b \rceil$, then $\Omega(n \log b/b)$ bits are necessary for external perfect hashing.

Sundar [47] considered the lower bound for the dynamic dictionary problem under the *refresh cost multilevel hashing model*. The result we are particularly interested in is the lower bound obtained under the *single-level hashing model*, which is essentially the same as the cell probe model. More specifically, Sundar proved that in the cell probe model with cell size b *bits* and one free cell, if a dictionary with maximum size n must perform search in worst-case 1 lookup, then the amortized update cost is $\Omega(n/b)$. This lower bound holds as long as $|U| \geq 2r$ and $r \geq n$.

Pagh [37] considered the cell probe complexity of membership and perfect hashing. He showed that in the word probe model, any data structure using linear space and must probe at least three cells to answer a membership query in worse case. This confirms the optimality of FSK scheme.

6.2 The Limit of Buffering

Buffering is an important technique in the I/O model. The presence of a no-cost internal memory could change the problem dramatically, since it can be used as a buffer space to batch up insertions and write them to disk periodically, fully utilizing the parallelism within one I/O and reducing the amortized insertion cost. The abundant research in the area of I/O-efficient data structures has witnessed this phenomenon numerous times, where the insertion cost can be typically brought down to close to $O(1/b)$ I/Os. Examples include the simplest structures like stacks and queues, to more advanced ones such as the *buffer tree* [3] and *priority queues* [4, 12]. Many of these results hold as long as the buffer has just a constant number of blocks; some require a larger buffer of $\Theta(b)$ blocks (known as the *tall cache* assumption). Please see the book by Vitter [51] for a complete account of the power of buffering.

Therefore the natural question is, can we (or not) lower the insertion cost of a dynamic hash table by buffering without sacrificing its near-perfect query performance? A folklore conjecture is that the insertion cost must be $\Omega(1)$ I/Os if the query cost is required to be $O(1)$ I/Os.

Successful Queries The first attempt to solve the conjecture is [53], in which only successful queries were considered. The main result is as follows:

Theorem 7 *For any constant $c > 0$, suppose we insert a sequence of $n > \Omega\left(m \log u \cdot b^{2c}\right)$ random items into an initially empty hash table. If the total cost of these insertions is expected $n \cdot t_u$ I/Os, and the hash table is able to answer a successful query in expected average t_q I/Os at any time, then the following tradeoffs hold:*

1. *If $t_q \leq 1 + O(1/b^c)$ for any $c > 1$, then $t_u \geq 1 - O(1/b^{\frac{c-1}{4}})$;*
2. *If $t_q \leq 1 + O(1/b)$, then $t_u \geq \Omega(1)$;*
3. *If $t_q \leq 1 + O(1/b^c)$ for any $0 < c < 1$, then $t_u \geq \Omega(b^{c-1})$.*

Theorem 7 says that if $t_q \leq 1 + O(1/b^c)$ for any constant $c > 1$, buffering is essentially useless. However, if the query cost is relaxed to $t_q \leq 1 + O(1/b^c)$ for any constant $0 < c < 1$, a simple dynamic hash table that supports insertions in $t_u \leq O(b^{c-1}) = o(1)$ I/Os is presented (for block sizes $b = \Omega(\log^{1/c} \frac{n}{m})$), indicating that by only considering successful queries we can not obtain any stronger lower bound result. Above results are pictorially illustrated in Figure 4.

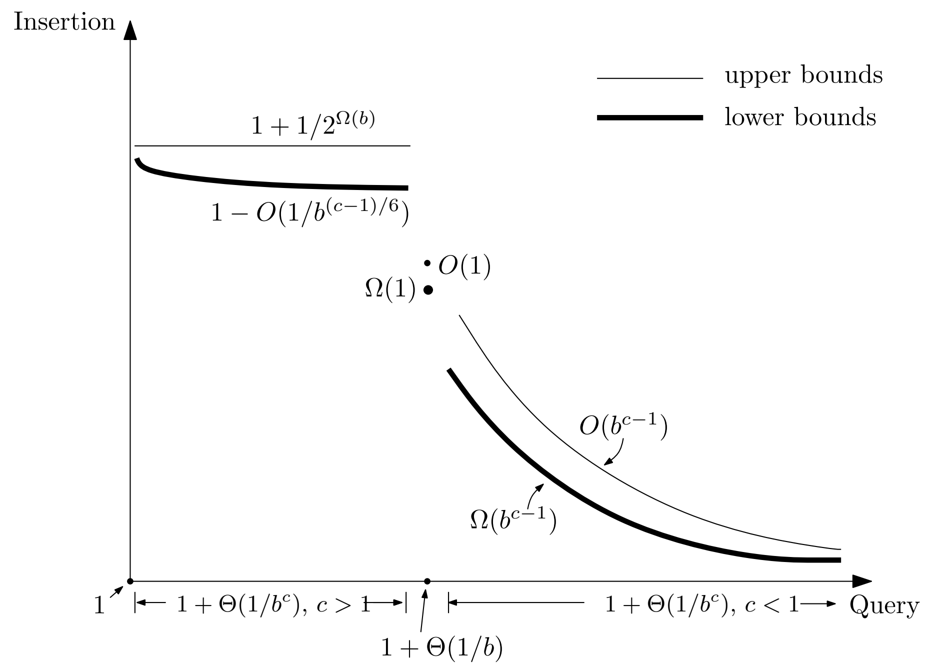

Figure 4. The query-insertion tradeoff.

Membership The lower bound in [53] can not be improved, since a matching upper bound for fast successful queries and efficient buffering is proposed. However for membership queries this is not the case. Zhang and Yi [55] shows that it is not possible to achieve $t_q = 1 + o(1)$ and $t_u = o(1)$ simultaneously. A very recent result by Verbin and Qin [49] shows that buffering is not possible for constant-time queries:

Theorem 8 *Suppose we insert a sequence of n elements chosen from a sufficiently large U uniformly at random, into any initially empty randomized data structure in the cell probe model with cell size b bits and free cells of total size m bits. If the amortized expected cost for a insertion is $t_u \leq 1 - O(1)$, then we must have $t_q \geq \Omega(\log_{b \log n}(n/m))$, where t_q is the expected number of probes made by data structure to answer a membership.*

7 Open Problems

Hashing is perhaps one of the most studied problems in computer science, and in this survey we can only cover a small subset of the results. It should be noted that after intensive study for decades, hashing remains a rich area and many questions remain open. Among them we find the following two particularly interesting and worth investigating:

1. How much randomness does cuckoo hashing need to achieve constant cost per operation? Cohen and Kane [8] shows that $\Omega(\log n)$-joint-independence is required, suggesting that it is possible that a lower bound of $\Omega(\log)$-wise independence exists.
2. How to design external hash tables using limited independence? Currently, all existing results use a truly random hash function. It would be nice to know how much randomness we need to achieve $1 + 1/2^{\Omega(b)}$ bounds.

References

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

[2] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority-queue and graph algorithms. In *Proc. ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[5] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 2000.

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

[7] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of the observations. *Ann. Math. Statist.*, 23:493–509, 1952.

[8] J. Cohen and D. Kane. Bounds on the Independence Required for Cuckoo Hashing. 2006. To appear.

[9] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Symposium on Theoretical Aspects of Computer Science*, pages 569–580, London, UK, 1996. Springer-Verlag.

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994.

[11] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

[12] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[13] 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.

[14] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20(4):345–369, 1983.

[15] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. In *Symposium on Theoretical Aspects of Computer Science*, pages 271–282, London, UK, 2003. Springer-Verlag.

[16] F. W. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 719–725, 1990.

[17] 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.

[18] 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.

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

[20] G. H. Gonnet and J. I. Munro. The analysis of linear probing sort by the use of a new mathematical transform. *Journal of Algorithms*, 5(4):451 – 470, 1984.

[21] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, 2001.

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

[23] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, Mar. 1963.

[24] D. E. Knuth. *Sorting and Searching*. Volume 3 of *The Art of Computer Programming* [25], second edition, 1998.

[25] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[26] P.-A. Larson. Linear hashing with partial expansions. In *Proc. International Conference on Very Large Data Bases*, pages 224–232. VLDB Endowment, 1980.

[27] P.-A. Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems*, 7(4):566–587, 1982.

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

[29] 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.

[30] H. G. Mairson. The program complexity of searching a table. In *SFCS '83: Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 40–47, Washington, DC, USA, 1983. IEEE Computer Society.

[31] K. Mehlhorn. On the program size of perfect and universal hash functions. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 170–175, Washington, DC, USA, 1982. IEEE Computer Society.

[32] 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.

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

[34] A. Pagh and R. Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.

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

[36] A. Pagh, R. Pagh, and M. Ružić. Linear probing with constant independence. *SIAM Journal on Computing*, 39(3):1107–1120, 2009.

[37] R. Pagh. On the cell probe complexity of membership and perfect hashing. In *Proc. ACM Symposium on Theory of Computing*, pages 425–432, 2001.

[38] R. Pagh and F. Rodler. Cuckoo hashing. *Proc. European Symposium on Algorithms*, pages 121–133.

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

[40] M. Pătraşcu and M. Thorup. On the k-independence required by linear probing and minwise independence. In *Proc. International Colloquium on Automata, Languages, and Programming*, 2010. To appear.

[41] K. Y. R Pagh, Z Wei and Q. Zhang. Cache-oblivious hashing. In *Proc. ACM Symposium on Principles of Database Systems*, 2010. To appear.

[42] J. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 224–234. ACM New York, NY, USA, 1990.

[43] J. P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness. In *Proc. ACM Symposium on Theory of Computing*, pages 224–234, New York, NY, USA, 1990. ACM.

[44] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited idependence. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 331–340, 1003.

[45] A. Siegel. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing*, 33(3):505–543, 2004.

[46] M. Smid. A worst-case algorithm for semi-online updates on decomposable problems. Report A-03/90, Dept. Comput. Sci., Univ. Saarlandes, Saarbrücken, West Germany, 1990.

[47] R. Sundar. A lower bound for the dictionary problem under a hashing model. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 612–621, 1991.

[48] M. Thorup. Even strongly universal hashing is pretty fast. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 496–497, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[49] 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.

[50] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[51] J. S. Vitter. *Algorithms and Data Structures for External Memory.* Now Publishers, 2008.

[52] M. Wegman and J. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

[53] 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.

[54] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

[55] K. Yi and Q. Zhang. On the cell probe complexity of dynamic membership. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2010.