Dynamic External Hashing: The Limit of Buffering*

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

Hash tables are one of the most fundamental data structures in computer science, in both theory and practice. They are especially useful in external memory, where their query performance approaches the ideal cost of just one disk access. Knuth [16] gave an elegant analysis showing that with some simple collision resolution strategies such as linear probing or chaining, the expected average number of disk I/Os of a lookup is merely $1 + 1/2^{\Omega(b)}$, where each I/O can read and/or write a disk block containing b items. Inserting a new item into the hash table also costs $1 + 1/2^{\Omega(b)}$ I/Os, which is again almost the best one can do if the hash table is entirely stored on disk. However, this requirement is unrealistic since any algorithm operating on an external hash table must have some internal memory (at least $\Omega(1)$ blocks) to work with. The availability of a small internal memory buffer can dramatically reduce the amortized insertion cost to $o(1)$ I/Os for many external memory data structures. In this paper we study the inherent query-insertion tradeoff of external hash tables in the presence of a memory buffer. In particular, we show that for any constant $c > 1$, if the expected average successful query cost is targeted at $1 + O(1/b^c)$ I/Os, then it is not possible to support insertions in less than $1 - O(1/b^{\frac{c-1}{6}})$ I/Os amortized, which means that the memory buffer is essentially useless. While if the query cost is relaxed to $1 + O(1/b^c)$ I/Os for any constant $c < 1$, there is a simple dynamic hash table with $o(1)$ insertion cost.

Categories and Subject Descriptors

F.2.3 [**Analysis of algorithms and problem complexity**]: Tradeoffs between complexity measures; E.2 [**Data storage**]: *hash-table representations*

General Terms

Theory

Keywords

Dynamic hash table, lower bound, successful query

1. INTRODUCTION

Hash tables are the most efficient way of searching for a particular item in a large database. They are arguably one of the most fundamental data structures in computer science, due to their simplicity of implementation, excellent performance in practice, and many nice theoretical properties. A hash table supports the following *dictionary* operations:

- *Insertion*: Insert a pair of (key, data) into the table;
- *Deletion*: Delete the (key, data) pair for a given key;
- *Successful query*: For a queried key that is present in the table, retrieve the data associated with it;
- *Unsuccessful query*: For a queried key that is not present in the table, return "not found".

A hash table supports all of the operations above in expected constant time. It works especially well in external memory, where the storage is divided into disk blocks, each containing up to b items. Thus collisions happen only when there are more than b items hashed to the same location. 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*, Knuth [16] showed that, under the ideal random hash function assumption, the expected average cost of a successful query is merely $1 + 1/2^{\Omega(b)}$ disk accesses (or *I/O*s), provided that the *load factor*[1] α is less than a constant smaller than 1. The expectation is with respect to the random choice of the hash function, while the average is with respect to the uniform choice of the queried item. An unsuccessful lookup also costs $1 + 1/2^{\Omega(b)}$ I/Os, but with a smaller constant in the big-Omega (i.e., slower). Knuth [16] gave an elegant analysis deriving the exact formula for the query cost, as a function of α and b. As typical values of b range from a few hundreds to a thousand, the query cost is extremely close to just one I/O.

Inserting an item into the hash table also costs $1 + 1/2^{\Omega(b)}$ I/Os: We simply insert the new item into the block where it

*Work of this paper was supported by Hong Kong Direct Allocation Grant (DAG 07/08). Qin Zhang was in addition supported by Hong Kong CERG Grant 613507.

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

[1]The load factor is defined to be ratio between the minimum number of blocks required to store n data records, $\lceil n/b \rceil$, and the actual number of blocks used by the hash table.

is supposed to go. If one wants to maintain the load factor we can periodically rebuild the hash table using schemes like *extensible hashing* [11] or *linear hashing* [17], but this only adds an extra cost of $O(1/b)$ I/Os amortized. Jensen and Pagh [15] demonstrate how to maintain the load factor at $\alpha = 1-O(1/b^{\frac{1}{2}})$ while still supporting queries in $1+O(1/b^{\frac{1}{2}})$ I/Os and updates in $1 + O(1/b^{\frac{1}{2}})$ I/Os. Indeed, one cannot hope for lower than 1 I/O for an insertion, if the hash table must reside on disk entirely and there is no space in main memory for buffering. However, this assumption is unrealistic, since any algorithm operating on an external data structure has to have at least a constant number of blocks of internal memory to work with. So we must include a main memory of size m in our setting to model the problem more accurately. In fact, this is exactly what the standard *external memory model* [1] depicts: The system has a disk of infinite size partitioned into blocks of size b, and a main memory of size $m > b$. Computation can only happen in main memory, which accesses the disk via I/Os. Each I/O can read and/or write a disk block storing up to b items, and the complexity is measured by the number of I/Os performed by the algorithm. Note that the model parameters m and b are considered asymptotic quantities. The presence of a no-cost main 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* [2] and *priority queues* [3, 10]. 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 [19] 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? Jensen and Pagh [15] recently conjectured that the insertion cost must be $\Omega(1)$ I/Os if the query cost is required to be $O(1)$ I/Os.

Our results. In this paper, we partially confirm the conjecture of Jensen and Pagh [15]. Specifically we obtain the following results. Consider any dynamic hash table that supports insertions in expected amortized t_u I/Os and answers a successful query in expected t_q I/Os on average. We show that if $t_q \leq 1+O(1/b^c)$ for any constant $c > 1$, then we must have $t_u \geq 1 - O(1/b^{\frac{c-1}{6}})$. This is only an additive term of $1/b^{\Omega(1)}$ away from how the standard hash table is supporting insertions, which means that buffering is essentially useless in this case. However, if the query cost is relaxed to $t_q \leq 1 + O(1/b^c)$ for any constant $0 < c < 1$, we present a simple dynamic hash table that supports insertions in $t_u = O(b^{c-1}) = o(1)$ I/Os (for block sizes $b = \Omega(\log^{1/c} \frac{n}{m})$). For this case we also present a matching lower bound of $t_u = \Omega(b^{c-1})$. Finally for the case $t_q = 1 + \Theta(1/b)$, we show a tight bound of $t_u = \Theta(1)$. Our results are pictorially illustrated in Figure 1, from which we see that we now have an almost complete understanding of the entire query-insertion tradeoff, and $t_q = 1 + \Theta(1/b)$ seems to be the sharp boundary separating effective and ineffective buffering. We prove our lower bounds for the three cases above using a unified framework in Section 2 and 3. The upper bound for the first case is simply the standard hash table following [16]; we give the upper bounds for the other two cases in Section 4.

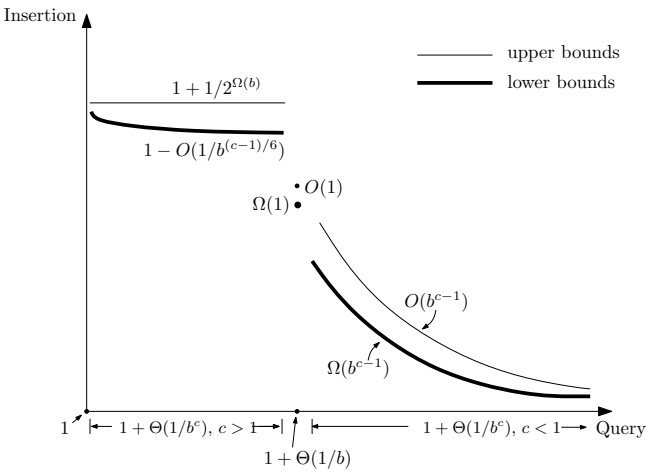

Figure 1. The query-insertion tradeoff.

Let $[u]$ be the universe of the keys. In Section 2, we prove that for any deterministic hash table, if we insert a total of n keys independently uniformly at random, there is a lower bound on the expected amortized cost per insertion, under the condition that at any time, the hash table must be able to answer a successful query with the desired expected average query bound. In Section 3 we show how this lower bound extends to randomized hash tables.

When proving our lower bounds we make the only requirement that items (i.e., the (key, data) pairs) must be treated as atomic elements, i.e., they can only be moved or copied between memory and disk in their entirety, and when answering a query, the query algorithm must visit the block (in memory or on disk) that actually contains the item or one of its copies. Such an *indivisibility* assumption is made in most external memory lower bounds, such as sorting, permuting [1], and all the range searching problems [4, 14, 21]. We assume that each machine word consists of $\log u$ bits and each item occupies one machine word (it does not affect our results if an item occupies any constant number of words). A block has b words and the memory stores up to m words. Finally, we comment that our lower bounds do not depend on the size of the hash table, which implies that the hash table cannot do better by consuming more disk space.

In this paper we only consider the tradeoff between successful query and insertion, since among the four types of dictionary operations, they are the two most important and common operations. In large databases, usually people do not do deletions at all due to the cheap storage, but just periodically rebuild the entire index [13]. For queries, the successful ones are much more common in many applications, since people use hash tables mainly for the purpose of retrieving the data associated with a key, not just testing if a key is present; better data structures exist for the latter purpose, such as *Bloom filters* [6]. We leave it as future work to consider other possible tradeoffs, for instance the trade-

off between max{successful query, unsuccessful query} and insertions, or between queries and max{insertion, deletion}.

Related results. Hash tables are widely used in practice due to their simplicity and excellent performance. Knuth's analysis [16] applies to the basic version where the hash table uses an ideal random hash function and t_q is the expected average cost. Afterward, a lot of works have been done to give better theoretical guarantees, for instance removing the ideal hash function assumption [8], making t_q worst-case [9, 12, 18], etc. Lower bounds have been sparse because in internal memory, the update time cannot be lower than $\Omega(1)$, which is already achieved by the standard hash table. Only with some strong requirements, e.g., when the algorithm is deterministic and t_q is worst-case, can one obtain some nontrivial lower bounds on the update time [9]. Our lower bounds, on the other hand, hold for randomized algorithms and do not need t_q to be worst-case.

As commented earlier, in external memory there is a trivial lower bound of 1 I/O for either a query or an update, if all the changes to the hash table must be committed to disk after each update. However, the vast amount of works in the area of external memory algorithms have never made such a requirement. And indeed for many problems, the availability of a small internal memory buffer can significantly reduce the amortized update cost without affecting the query cost [2, 3, 10, 19]. Unfortunately, little is known on the inherent limit of what buffering can do. The only nontrivial lower bounds on the update cost of any external data structure with a memory buffer are a paper by Fagerberg and Brodal [7] on the *predecessor* problem and a recent result of Yi [21] on the *range reporting* problem. But the techniques used are inapplicable to our problem. To the best of our knowledge, no nontrivial lower bound on external hashing of any kind is known.

2. LOWER BOUNDS

In this section, we prove a lower bound for any deterministic hash table under a total of n independent and random insertions, for some sufficiently large n. We will derive a lower bound on t_u, the expected amortized number of I/Os for an insertion, while assuming that the hash table is able to answer a successful query in t_q I/Os on average in expectation after the first i items have been inserted, for all $i = 1, \ldots, n$. We assume that all the keys are different, which happens with probability $1 - O(1/n)$ as long as $u > n^3$ by the birthday paradox. Under this setting we obtain the following tradeoffs between t_q and t_u.

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

The abstraction. To abstractly model a dynamic hash table, we ignore any of its auxiliary structures but only focus on the layout of items. Consider any snapshot of the hash table when we have inserted k items. We divide these k items into three zones. The *memory zone* M is a set of at most m items that are kept in memory. It takes no I/O to query any item in M. All items not in M must reside on disk. Denote all the blocks on disk by $B_1, B_2, \ldots, B_d$. Each B_i is a set of at most b items, and it is possible that one item appears in more than one B_i. Let $f : U \to \{1, \ldots, d\}$ be any function computable within memory, and we divide the disk-resident items into two zones with respect to f and the set of blocks $B_1, \ldots, B_d$. The *fast zone* F contains all items x such that $x \in B_{f(x)}$: These are the items that are accessible with just one I/O. We allocate all the remaining items into the *slow zone* S: These items need at least two I/Os to locate. Note that under random inputs, the sets $M, F, S, B_1, \ldots, B_d$ are all random sets, which the hash table will adaptively choose after seeing each random insertion. Changing M is free, but changing any B_i will cost 1 I/O.

Any query algorithm on the hash table can be modeled as described, since the only way to find a queried item in one I/O is to compute the index of a block containing x with only the information in memory. If the memory-resident computation gives an incorrect address or anything else, at least 2 I/Os will be necessary. Because any such f must be computable within memory, and the memory has $m \log u$ bits, the hash table can employ a family $\mathcal{F}$ of at most $2^{m \log u}$ distinct f's. Note that the current f adopted by the hash table is dependent upon the already inserted items, but the family $\mathcal{F}$ has to be fixed beforehand.

Suppose the hash table answers a successful query with an expected average cost of $t_q = 1 + \delta$ I/Os, where $\delta = 1/b^c$ for some constant $c > 0$. Consider the snapshot of the hash table when k items have been inserted. Then we must have $\mathbf{E}[|F| + 2 \cdot |S|]/k \le 1 + \delta$. Since $|F| + |S| = k - |M|$ and $\mathbf{E}[|M|] \le m$, we have

$$\mathbf{E}[|S|] \le m + \delta k. \tag{1}$$

We also have the following high-probability version of (1).

LEMMA 1. *Let $\phi \ge 1/b^{(c-1)/4}$ and let $k \ge \phi n$. At the snapshot when k items have been inserted, with probability at least $1 - 2\phi$, $|S| \le m + \frac{\delta}{\phi} k$.*

PROOF. On this snapshot the hash table answers a query in expected average $1+\delta$ I/Os. We claim that with probability at most 2ϕ, the average query cost is more than $1 + \delta/\phi$. Otherwise, since in any case the average query cost is at least $1 - m/k$ (assuming all items not in memory need just one I/O), we would have an expected average cost of at least

$$(1 - 2\phi)(1 - m/k) + 2\phi \cdot (1 + \delta/\phi) > 1 + \delta,$$

provided that $\frac{n}{m} > \frac{1}{\phi\delta}$, which is valid since we assume that $\frac{n}{m} > b^{2c} \log u$. The lemma then follows from the same argument used to derive (1). □

Basic idea of the lower bound proof. For the first ϕn items, we ignore the cost of their insertions. Consider any $f : U \to \{1, \ldots, d\}$. For $i = 1, \ldots, d$, let $\alpha_i = |f^{-1}(i)|/u$, and we call $(\alpha_1, \ldots, \alpha_d)$ the *characteristic vector* of f. Note that $\sum_i \alpha_i = 1$. For any one of the first ϕn items, since it is randomly chosen from U, f will direct it to B_i with

probability α_i. Intuitively, if α_i is large, too many items will be directed to B_i. Since B_i contains at most b items, the extra items will have to be pushed to the slow zone. If there are too many large α_i's, S will be large enough to violate the query requirement. Thus, the hash table should use an f that distributes items relatively evenly to the blocks. However, if f evenly distributes the first ϕn items, it is also likely to distribute newly inserted items evenly, leading to a high insertion cost. Below we formalize this intuition.

For the first tradeoff of Theorem 1, we set $\delta = 1/b^c$. We also pick the following set of parameters $\phi = 1/b^{(c-1)/4}, \rho = 2b^{(c+3)/4}/n, s = n/b^{(c+1)/2}$. We will use different values for these parameters when proving the other two tradeoffs. Given an f with characteristic vector $(\alpha_1, \ldots, \alpha_d)$, let $D^f = \{i \mid \alpha_i > \rho\}$ be the collection of block indices with large α_i's. We say that the indices in D^f form the *bad index area* and others form the *good index area*. Let $\lambda_f = \sum_{i \in D^f} \alpha_i$. Note that there are at most λ_f/ρ indices in the bad index area. We call an f with $\lambda_f > \phi$ a *bad function*; otherwise it is a *good function*. The following lemma shows that with high probability, the hash table has to use a good function f from $\mathcal{F}$.

LEMMA 2. *At the snapshot when k items are inserted for any $k \geq \phi n$, the function f used by the hash table is a good function with probability at least $1 - 2\phi - 1/2^{\Omega(b)}$.*

PROOF. Consider any bad function f from $\mathcal{F}$. Let X_j be the indicator variable of the event that the j-th inserted item is mapped to the bad index area, $j = 1, \ldots, k$. Then $X = \sum_{j=1}^{k} X_j$ is the total number of items mapped to the bad index area of f. We have $\mathbf{E}[X] = \lambda_f k$. By the Chernoff bound, we have

$$\Pr\left[X < \frac{2}{3}\lambda_f k\right] \leq e^{-\frac{(1/3)^2 \lambda_f k}{2}} \leq e^{-\frac{\phi^2 n}{18}},$$

namely with probability at least $1 - e^{-\frac{\phi^2 n}{18}}$, we have $X \geq \frac{2}{3}\lambda_f k$. Since the family $\mathcal{F}$ contains at most $2^{m \log u}$ bad functions, by a union bound we know that with probability at least $1 - 2^{m \log u} \cdot e^{-\frac{\phi^2 n}{18}} \geq 1 - 1/2^{\Omega(b)}$ (by the parameters chosen and the assumption that $n > \Omega(mb^{2c} \log u)$), for all the bad functions in $\mathcal{F}$, we have $X \geq \frac{2}{3}\lambda_f k$.

Consequently, since the bad index area can only accommodate $b \cdot \lambda_f/\rho$ items in the fast zone, at least $\frac{2}{3}\lambda_f k - b\lambda_f/\rho$ cannot be in the fast zone. The memory zone can accept at most m items, so the number of items in the slow zone is at least

$$|S| \geq \frac{2}{3}\lambda_f k - b\lambda_f/\rho - m > m + \frac{\delta}{\phi}k.$$

This happens with probability at least $1 - 1/2^{\Omega(b)}$, due to the fact that f is a bad function. On the other hand, Lemma 1 states that $|S| \leq m + \frac{\delta}{\phi}k$ holds with probability at least $1 - 2\phi$, thus by a union bound the hash table has to use a good function with probability at least $1 - 2\phi - 1/2^{\Omega(b)}$. □

A bin-ball game. Lemma 2 enables us to consider only those good functions f after the initial ϕn insertions. To show that any good function will incur a large insertion cost, we first consider the following bin-ball game, which captures the essence of performing insertions using a good function.

In an (s, p, t) *bin-ball game*, we throw s balls into r (for any $r \geq 1/p$) bins independently at random, and the probability that any ball goes to any particular bin is no more than p. At the end of the game, an adversary removes t balls from the bins such that the remaining $s - t$ balls hit the least number of bins. The cost of the game is defined as the number of nonempty bins occupied by the $s - t$ remaining balls.

We have the following two results with respect to such a game, depending on the relationships among s, p, and t.

LEMMA 3. *If $sp \leq \frac{1}{3}$, then for any $\mu > 0$, with probability at least $1 - e^{-\frac{\mu^2 s}{3}}$, the cost of an (s, p, t) bin-ball game is at least $(1 - \mu)(1 - sp)s - t$.*

PROOF. Imagine that we throw the s balls one by one. Let X_j be the indicator variable denoting the event that the j-th ball is thrown into an empty bin. The number of nonempty bins in the end is thus $X = \sum_{j=1}^{s} X_j$. These X_j's are not independent, but no matter what has happened previously for the first $j - 1$ balls, we always have $\Pr[X_j = 0] \leq sp$. This is because at any time, at most s bins are nonempty. Let Y_j $(1 \leq j \leq s)$ be a set of independent variables such that

$$Y_i = \begin{cases} 0, & \text{with probability } sp; \\ 1, & \text{otherwise.} \end{cases}$$

Let $Y = \sum_{j=1}^{s} Y_j$. Each Y_i is stochastically dominated by X_i, so Y is stochastically dominated by X. We have $\mathbf{E}[Y] = (1 - sp)s$ and we can apply the Chernoff bound on Y:

$$\Pr\left[Y < (1 - \mu)(1 - sp)s\right] < e^{-\frac{\mu^2 (1 - sp)s}{2}} < e^{-\frac{\mu^2 s}{3}}.$$

Therefore with probability at least $1 - e^{-\frac{\mu^2 s}{3}}$, we have $X \geq (1 - \mu)(1 - sp)s$. Finally, since removing t balls will reduce the number of nonempty bins by at most t, the cost of the bin-ball game is at least $(1 - \mu)(1 - sp)s - t$ with probability at least $1 - e^{-\frac{\mu^2 s}{3}}$. □

LEMMA 4. *If $s/2 \geq t$ and $s/2 \geq 1/p$, then with probability at least $1 - 1/2^{\Omega(s)}$, the cost of an (s, p, t) bin-ball game is at least $1/(20p)$.*

PROOF. In this case, the adversary will remove at most $s/2$ balls in the end. Thus we show that with very small probability, there exist a subset of $s/2$ balls all of which are thrown into a subset of at most $1/(20p)$ bins. Before the analysis, we merge bins such that the probability that any ball goes to any particular bin is between $p/2$ and p, and consequently, the number of of bins would be between $1/p$ to $2/p$. Note that such an operation will only make the cost of the bin-ball game smaller. Now this probability is at most

$$\sum_{i=1}^{1/(20p)} \left(\binom{2/p}{i} \binom{s}{s/2} \left(\frac{i}{1/p}\right)^{s/2} \right) \leq$$

$$\leq \quad 2\binom{2/p}{1/(20p)} \binom{s}{s/2} \left(\frac{1/(20p)}{1/p}\right)^{s/2} \leq 1/2^{\Omega(s)},$$

hence the lemma. □

Now we are ready to prove the main theorem.

Proof of Theorem 1.

PROOF. We begin with the first tradeoff. Recall that we use the following parameters: $\delta = 1/b^c$, $\phi = 1/b^{(c-1)/4}$, $\rho = 2b^{(c+3)/4}/n$, $s = n/b^{(c+1)/2}$. For the first ϕn items, we do not count their insertion costs. We divide the rest of the insertions into rounds, with each round containing s items. We now bound the expected cost of each round.

Focus on a particular round, and let f be the function used by the hash table at the end of this round. We only consider the set R of items inserted in this round that are mapped to the good index area of f, i.e., $R = \{x \mid f(x) \notin D^f\}$; other items are assumed to have been inserted for free. Consider the block with index $f(x)$ for a particular x. If x is in the fast zone, the block $B_{f(x)}$ must contain x. Thus, the number of distinct indices $f(x)$ for $x \in R \cap F$ is an obvious lower bound on the I/O cost of this round. Denote this number by $Z = |\{f(x) \mid x \in R \cap F\}|$. Below we will show that Z is large with high probability.

We first argue that at the end of this round, each of the following three events happens with high probability.

- $\mathcal{E}_1$: $|S| \le \delta n/\phi + m$;
- $\mathcal{E}_2$: f is a good function;
- $\mathcal{E}_3$: For all good functions $f \in \mathcal{F}$ and their corresponding slow zones S and memory zones M, $Z \ge (1 - O(\phi))s - t$, where $t = |S| + |M|$.

By Lemma 1, $\mathcal{E}_1$ happens with probability at least $1 - 2\phi$. By Lemma 2, $\mathcal{E}_2$ happens with probability at least $1 - 2\phi - 1/2^{\Omega(b)}$. It remains to show that $\mathcal{E}_3$ also happens with high probability.

We prove so by first claiming that for a particular good function $f \in \mathcal{F}$, with probability at least $1 - e^{-2\phi^2 s}$, Z is at least the cost of a $((1-2\phi)s, \frac{\rho}{1-\lambda_f}, t)$ bin-ball game. This is because of the following reasons:

1. Since f is a good function, by the Chernoff bound, with probability at least $1 - e^{-2\phi^2 s}$, more than $(1-2\phi)s$ newly inserted items will fall into the good index area of f, i.e., $|R| > (1-2\phi)s$.
2. The probability of any item being mapped to any index in the good index area, conditioned on that it goes to the good index area, is no more than $\frac{\rho}{1-\lambda_f}$.
3. Only t items in R are not in the fast zone F, excluding them from R corresponds to discarding t balls at the end of the bin-ball game.

Thus by Lemma 3 (setting $\mu = \phi$), with probability at least $1 - e^{-\frac{\phi^2 \cdot (1-2\phi)s}{3}} - e^{-2\phi^2 s}$, we have

$$\begin{aligned} Z &\ge (1-\phi)\left(1 - (1-2\phi)s \cdot \frac{\rho}{1-\lambda_f}\right)(1-2\phi)s - t \\ &\ge (1-\phi)\left(1 - (1-2\phi)s \cdot \frac{\rho}{1-\phi}\right)(1-2\phi)s - t \\ &\ge (1 - O(\phi))\, s - t. \end{aligned}$$

Thus by applying a union bound on all good functions in $\mathcal{F}$, $\mathcal{E}_3$ happens with probability at least $1 - (e^{-\frac{\phi^2 \cdot (1-2\phi)s}{3}} + e^{-2\phi^2 s}) \cdot 2^{m \log u} = 1 - 2^{-\Omega(b)}$ (by the assumption $n > \Omega(mb^{2c} \log u)$).

Now we lower bound the expected insertion cost of one round. By a union bound, with probability at least $1 - O(\phi) - 1/2^{\Omega(b)}$, all of $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_3$ happen at the end of the round. By $\mathcal{E}_2$ and $\mathcal{E}_3$, we have $Z \ge (1 - O(\phi))\, s - t$. Since now $t = |S| + |M| \le \delta n/\phi + 2m = O(\phi s)$ by $\mathcal{E}_1$, we have $Z \ge (1 - O(\phi))\, s$. Thus the expected cost of one round will be at least

$$(1 - O(\phi))\, s \cdot \left(1 - O(\phi) - 1/2^{\Omega(b)}\right) = (1 - O(\phi))\, s.$$

Finally, since there are $(1-\phi)n/s$ rounds, the expected amortized cost per insertion is at least

$$(1 - O(\phi))\, s \cdot (1-\phi)n/s \cdot 1/n = 1 - O\left(1/b^{\frac{c-1}{4}}\right).$$

For the second tradeoff, we choose the following set of parameters: $\phi = 1/\kappa, \rho = 2\kappa b/n, s = n/(\kappa^2 b)$ and $\delta = 1/(\kappa^4 b)$ (for some constant κ large enough). We can check that Lemma 2 still holds with these parameters, and then go through the proof above. We omit the tedious details. Plugging the new parameters into the derivations we obtain a lower bound $t_u \ge \Omega(1)$.

For the third tradeoff, we choose the following set of parameters: $\phi = 1/8, \rho = 16b/n, s = 32n/b^c$ and $\delta = 1/b^c$. We can still check the validity of Lemma 2, and go through the whole proof. The only difference is that we need to use Lemma 4 in place of Lemma 3, the reason being that for our new set of parameters, we have $s\rho = \omega(1)$ thus Lemma 3 does not apply. By using Lemma 4 we can lower bound the expected insertion cost of each round by $\Omega\left((1-2\phi)/(20\rho)\right)$, so the expected amortized insertion cost is at least

$$\Omega\left(\frac{1-2\phi}{20\rho}\right) \cdot (1-\phi)n/s \cdot 1/n = \Omega(b^{c-1}),$$

as claimed. □

3. LOWER BOUNDS FOR RANDOMIZED HASH TABLES

In this section we show how to extend our lower bound to randomized hash tables. We follow the framework of Yao [20]. A randomized hash table can be viewed as a probability distribution $P_{\mathcal{A}}$ over the set $\mathcal{A}$ of all deterministic hash tables. We still consider the tradeoff between the expected average cost of a successful query t_q and the expected amortized insertion cost t_u. Now the expectation is with respect to the probability distribution $P_{\mathcal{A}}$. More precisely, let $Q(A, I, t)$ denote the average query cost over all items in the deterministic hash table $A \in \mathcal{A}$, on input sequence I at snapshot t, and $U(A, I)$ denote the I/O cost per item of inserting all items in I using A, then the expected average query cost and expected amortized insertion cost of a randomized hash table $P_{\mathcal{A}}$ can be expressed as $t_q = \max_I \max_t \mathbf{E}_{P_{\mathcal{A}}}[Q(A, I, t)]$ and $t_u = \max_I \mathbf{E}_{P_{\mathcal{A}}}[U(A, I)]$, respectively. For randomized hash tables we have the following tradeoffs:

THEOREM 2. *For any randomized hash table, suppose we insert a sequence of $n > \Omega\left(m \log u \cdot b^{2c}\right)$ items into it. If the total cost of these insertions is expected at most $n \cdot t_u$ I/Os under any input, 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* $t_q \le 1 + O(1/b^c)$ *for any* $c > 1$, *then* $t_u \ge 1 - O(1/b^{\frac{c-1}{6}})$;

2. *If* $t_q \le 1 + O(1/b)$, *then* $t_u \ge \Omega(1)$;

3. *If* $t_q \le 1 + O(1/b^c)$ *for any* $0 < c < 1$, *then* $t_u \ge \Omega(b^{c-1})$.

PROOF. For the first tradeoff, we set the parameters as follows: $\phi = 1/b^{(c-1)/6}, \rho = 2b^{(c+5)/6}, s = n/b^{(c+2)/3}$. Assuming the query cost

$$\max_I \max_t \mathbf{E}_{P_\mathcal{A}}[Q(A, I, t)] \le 1 + O(1/b^c)$$

for any $c > 1$, we will derive a lower bound for the insertion cost $\max_I \mathbf{E}_{P_\mathcal{A}}[Q(A, I)]$. Let $P_\mathcal{I}$ denote the uniform distribution over the set $\mathcal{I}$ of all input sequences of length n. Considering $\frac{1}{n}\sum_{t=1}^{n} \mathbf{E}_{P_\mathcal{I}, P_\mathcal{A}}[Q(A, I, t)]$, we have the following bound:

$$\begin{aligned}
\frac{1}{n}\sum_{t=1}^{n} \mathbf{E}_{P_\mathcal{I}, P_\mathcal{A}}[Q(A, I, t)] &= \frac{1}{n}\sum_{t=1}^{n} \mathbf{E}_{P_\mathcal{I}}\left[\mathbf{E}_{P_\mathcal{A}}[Q(A, I, t)]\right] \\
&\le \frac{1}{n}\sum_{t=1}^{n} \mathbf{E}_{P_\mathcal{I}}[1 + O(1/b^c)] \\
&\le 1 + O(1/b^c).
\end{aligned}$$

Consider any $t \ge \phi n$. Since $Q(A, I, t) \ge 1 - m/\phi n$ for all A and I, it follows that with probability at least $1 - \phi$, the (deterministic) hash table A chosen according to $P_\mathcal{A}$ satisfies

$$\frac{1}{n}\sum_{t=1}^{n} \mathbf{E}_{P_\mathcal{I}}[Q(A, I, t)] \le 1 + O\left(\frac{1}{\phi b^c}\right),$$

by the parameters chosen above and for n large enough. We will prove that for these hash tables, the insertion cost is large. Fixing such a hash table A, it is easy to show that A satisfies $\mathbf{E}_{P_\mathcal{I}}[Q(A, I, t)] \le 1 + O\left(\frac{1}{\phi^2 b^c}\right)$ on at least $(1-2\phi)n$ snapshots. Call these snapshots *good snapshots*. We can check that Lemma 1 and Lemma 2 still hold on any good snapshot. Ignoring the first ϕn insertions, we divide the remaining $(1 - \phi)n$ insertions into rounds, with each round containing exactly s good snapshots and also ending with a good one. Focusing on a particular round, we only consider the insertion cost of the s items inserted right before the good snapshots. Since the ending snapshot of the round is good, using the same argument as in Theorem 1 we can prove that the cost inserting the s items is at least $(1-O(\phi))s$. So the total insertion cost of each round is at least $(1-O(\phi))s$. Since there are $(1 - 2\phi)n/s$ rounds, the expected amortized cost per insertion of A is $\mathbf{E}_{P_\mathcal{I}}[U(A, I)] \ge (1 - O(\phi))s \cdot (1 - 2\phi)n/s \cdot 1/n = 1 - O(\phi)$. For the randomized hash table $P_\mathcal{A}$, since with probability $\ge 1 - \phi$, A is one for which the above analysis goes through, we can bound the expected amortized insertion cost as follows:

$$\begin{aligned}
\max_I \mathbf{E}_{P_\mathcal{A}}[U(A, I)] &\ge \mathbf{E}_{P_\mathcal{A}, P_\mathcal{I}}[U(A, I)] \\
&\ge (1 - \phi)(1 - O(\phi)) \\
&\ge 1 - O(1/b^{\frac{c-1}{6}}).
\end{aligned}$$

For the second and third tradeoffs, we choose the same parameters as in the proof of Theorem 1, and a similar argument will yield the desired results. □

4. UPPER BOUNDS

In this section, we present some upper bounds on the query-insertion tradeoff of external hash tables, showing that all three lower bound tradeoffs of Theorem 1 are essentially tight. The first tradeoff is matched by the standard external hash table, up to an additive term of $1/b^{\Omega(1)}$. Below we give matching (up to constant factors) upper bounds for the other two tradeoffs.

Specifically, we present a simple dynamic hash table that, for any constant $0 < c \le 1$, supports insertions in $t_u = O(b^{c-1} + \frac{\log(n/m)}{b})$ I/Os amortized, while being able to answer a query in expected $t_q = 1 + O(1/b^c)$ I/Os on average. This means that our lower bound is tight for all block sizes $b = \Omega(\log^{1/c} \frac{n}{m})$. In the following we first state a folklore result by applying the *logarithmic method* [5] to a standard hash table [16], then we show how to improve the query cost to $1 + O(1/b^c)$ while keeping the insertion cost low.

Applying the logarithmic method. Fix a parameter $\gamma \ge 2$. We maintain a series of hash tables $\mathcal{H}_0, \mathcal{H}_1, \ldots$. The hash table $\mathcal{H}_k$ has $\gamma^k \cdot \frac{m}{b}$ buckets and stores up to $\frac{1}{2}\gamma^k m$ items, so that its load factor is always at most $\frac{1}{2}$. We use some standard method to resolve collisions, such as chaining. The first hash table $\mathcal{H}_0$ always resides in memory while the rest stay on disk.

When a new item is inserted, it always goes to the memory-resident $\mathcal{H}_0$. When $\mathcal{H}_0$ is full (i.e., having $\frac{1}{2}m$ items), we migrate all items stored in $\mathcal{H}_0$ to $\mathcal{H}_1$. If $\mathcal{H}_1$ is not empty, we simply merge the corresponding buckets. Note that each bucket in $\mathcal{H}_0$ corresponds to γ consecutive buckets in $\mathcal{H}_1$, and we can easily distribute the items to their new buckets in $\mathcal{H}_1$ by scanning the two tables in parallel, costing $O(\gamma \cdot \frac{m}{b})$ I/Os. This operation takes place inductively: Whenever $\mathcal{H}_k$ is full, we migrate its items to $\mathcal{H}_{k+1}$, costing $O(\gamma^{k+1} \cdot \frac{m}{b})$ I/Os. Then standard analysis shows that for n insertions, the total cost is $O(\frac{\gamma n}{b} \log \frac{n}{m})$ I/Os, or $O(\frac{\gamma}{b} \log \frac{n}{m})$ amortized I/Os per insertion. However, for a query we need to examine all the $O(\log_\gamma \frac{n}{m})$ hash tables.

LEMMA 5. *For any parameter* $\gamma \ge 2$, *there is a dynamic hash table that supports an insertion in amortized* $O(\frac{\gamma}{b} \log \frac{n}{m})$ *I/Os and a successful query in expected* $O(\log_\gamma \frac{n}{m})$ *I/Os.*

Improving the query cost. Next we show how to improve the average cost of a successful query to $1+O(1/b^c)$ I/Os for any constant $0 < c \le 1$, while keeping the insertion cost low. The idea is to try to put the majority of the items into one single big hash table. In the standard logarithmic method described above, the last table may seem a good candidate, but sometimes it may only contain a constant fraction of all items. Below we show how to bootstrap the structure above to obtain a better query bound.

Fix a parameter $2 \le \beta \le b$. For the first m items inserted, we dump them in a hash table $\widehat{\mathcal{H}}$ on disk. Then run the algorithm of Lemma 5 for the next m/β items. After that we merge these m/β items into $\widehat{\mathcal{H}}$. We keep doing so until the size of $\widehat{\mathcal{H}}$ has reached $2m$, and then we start the next round. Generally, in the i-th round, the size of $\widehat{\mathcal{H}}$ goes from $2^{i-1}m$ to $2^i m$, and we apply the algorithm of Lemma 5 for every $2^{i-1}m/\beta$ items. It is clear that $\widehat{\mathcal{H}}$ always has at least a fraction of $1 - \frac{1}{\beta}$ of all the items inserted so far, while

the series of hash tables used in the logarithmic method maintain at least a separation factor of 2 in the sizes between successive tables. Thus, the expected average query cost is at most

$$\left(1 + 1/2^{\Omega(b)}\right)\left(1 \cdot \left(1 - \frac{1}{\beta}\right) + \frac{1}{\beta}\left(2 \cdot \frac{1}{2} + 3 \cdot \frac{1}{4} + \cdots\right)\right)$$

$= 1 + O(1/\beta)$.

Next we analyze the amortized insertion cost. Since the number of items doubles every round, it is (asymptotically) sufficient to analyze the last round. In the last round, $\widehat{\mathcal{H}}$ is scanned β times, and we charge $O(\beta/b)$ I/Os to each of the n items. The algorithm of Lemma 5 is invoked β times, but every invocation handles $O(n/\beta)$ different items, so the amortized cost per item is still $O(\frac{\gamma}{b}\log\frac{n}{m})$ I/Os. Thus the total amortized cost per insertion is $O(\frac{1}{b}(\beta+\gamma\log\frac{n}{m}))$ I/Os. Then setting $\beta = b^c$ and $\gamma = 2$ yields the desired results.

THEOREM 3. *For any constant $0 < c \leq 1$, there is a dynamic hash table that supports an insertion in amortized $O(b^{c-1} + \frac{\log(n/m)}{b})$ I/Os and a successful query in expected average $1 + O(1/b^c)$ I/Os.*

5. FINAL REMARKS

It is clear that the hash table of Theorem 3 is optimized for the average cost of successful queries, but has a rather poor performance on unsuccessful queries. The conjecture in [15] is that buffering should be entirely useless if both query costs are to be $O(1)$.

Nevertheless, even for successful queries the problem is still not completely understood. There is a gap between our upper and lower bound for smaller block sizes $b = O(\log^{1/c}\frac{n}{m})$ and it will be interesting to see how small the blocks can be so as to still allow for effective buffering. Note that when $b = O(1)$, the external memory model essentially becomes RAM, and buffering certainly will not help in this case.

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

[3] 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 Computation*, pages 268–276, 2002.

[4] L. Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. *Algorithmica*, 54(3), 2009.

[5] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13, pages 422–426, 1970.

[7] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

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

[9] 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.

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

[11] 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.

[12] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[13] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[14] J. M. Hellerstein, E. Koutsoupias, D. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM*, 49(1):35–55, 2002.

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

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

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

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

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

[20] A. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. IEEE Symposium on Foundations of Computer Science*, 1977.

[21] K. Yi. Dynamic indexability and lower bounds for dynamic one-dimensional range query indexes. In *Proc. ACM Symposium on Principles of Database Systems*, 2009.