Optimal Hashing in External Memory

Alex Conway
Rutgers University, New Brunswick, NJ, USA
alexander.conway@rutgers.edu

Martín Farach-Colton
Rutgers University, New Brunswick, NJ, USA
farach@rutgers.edu

Philip Shilane
Dell EMC, Newtown, PA, USA
shilane@dell.com

Abstract

Hash tables are a ubiquitous class of dictionary data structures. However, standard hash table implementations do not translate well into the external memory model, because they do not incorporate locality for insertions.

Iacono and Pătraşu established an update/query tradeoff curve for external-hash tables: a hash table that performs insertions in $O(\lambda/B)$ amortized IOs requires $\Omega(\log_\lambda N)$ expected IOs for queries, where N is the number of items that can be stored in the data structure, B is the size of a memory transfer, M is the size of memory, and λ is a tuning parameter. They provide a complicated hashing data structure, which we call the **IP hash table**, that meets this curve for λ that is $\Omega(\log\log M + \log_M N)$.

In this paper, we present a simpler external-memory hash table, the **Bundle of Arrays Hash Table** (BOA), that is optimal for a narrower range of λ. The simplicity of BOAs allows them to be readily modified to achieve the following results:

- A new external-memory data structure, the **Bundle of Trees Hash Table** (BOT), that matches the performance of the IP hash table, while retaining some of the simplicity of the BOAs.
- The **Cache-Oblivious Bundle of Trees Hash Table** (COBOT), the first cache-oblivious hash table. This data structure matches the optimality of BOTs and IP hash tables over the same range of λ.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases hash tables, external memory algorthims, cache-oblivious algorithms, asymmetric data structures

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.39

Related Version A full version of the paper is available at [?], `http://arxiv.org/abs/1805.09423`.

Funding Supported by NSF CCF 1637458, NIH 1 U01 CA198952-01, a NetAPP Faculty Fellowship and a gift from Dell/EMC.

1 Introduction

Dictionaries are among the most heavily used data structures. A dictionary maintains a collection of key-value pairs $\mathcal{S} \subseteq \mathcal{U} \times \mathcal{V}$, under operations[1] $\textsc{insert}(x, v, \mathcal{S})$, $\textsc{delete}(x, \mathcal{S})$, and $\textsc{query}(x, \mathcal{S})$, which returns the value corresponding to x when $x \in S$. When data fits in memory, there are many solutions to the dictionary problem.

When data is too large to fit in memory, comparison-based dictionaries can be quite varied. They include the B^{ε}-tree [8], the write-optimized skip list [6], and the cache-optimized look-ahead array (COLA) [2,3,5]. These are optimal in the **external-memory comparison model** in that they match the bound established by Brodal and Fagerberg [8] who showed that for any dictionary in this model, if insertions can be performed in $O\left(\frac{\lambda \log_\lambda N}{B}\right)$ amortized IOs, then there exists a query that requires at least $\Omega(\log_\lambda N)$ IOs, where N is the number of items that can be stored in the data structure, B is the size of a memory transfer, and λ is a tuning parameter. In the following M will be the size of memory, and $B = \Omega(\log n)$. This trade off has since been extended in several ways [1, 4].

Iacono and Pătraşcu showed that in the DAM model, in which operations beyond comparisons are allowed on keys, that a better tradeoff exists:

▶ **Theorem 1** ([11]). *If insertions into an external memory dictionary can be performed in $O(\lambda/B)$ amortized IOs, then queries require an expected $\Omega(\log_\lambda N)$ IOs.*

They further describe an external-memory hashing algorithm, which we refer to here as the **IP hash table**, that performs insertions in $O\left(\frac{1}{B}\left(\lambda + \log_{\frac{M}{B}} N + \log\log N\right)\right)$ IOs and queries in $O(\log_\lambda N)$ IOs w.h.p. Therefore, for $\lambda = \Omega\left(\log_{M/B} N + \log\log N\right)$, the IP hash table meets the tradeoff curve of Theorem 1 and is thus optimal.

In dictionaries that do not support successors and predecessors, we can assume that keys are hashed, that is, that they are uniformly distributed and satisfy some independence properties. The IP hash table and the following results hash all keys before insertion and query in the dictionary by a $\Theta(\log N)$-independent hash function.

The base result of this paper is a simple external-memory hashing scheme, the **Bundle of Arrays Hash Table** (BOA), that meets the optimal Theorem 1 trade off curve for large enough λ. Specifically, we show:

▶ **Theorem 2.** *A BOA supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log_\lambda N\right)/B\right)$ IOs, for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

Thus BOAs are optimal for $\lambda = \Omega(\log_{\frac{M}{B}} N + \log_\lambda N)$. They are readily modified to provide several variations, notably the **Bundle of Trees Hash Table** (BOT). BOTs are optimal for the same range of λ as the IP hash table:

▶ **Theorem 3.** *A BOT supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log\log M\right)/B\right)$ IOs for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

[1] We do not consider dictionaries that also support the $\textsc{succ}(x, \mathcal{S})$ and $\textsc{pred}(x, \mathcal{S})$. $\textsc{succ}(x, \mathcal{S})$ return $\min\{y | y > x \wedge y \in \mathcal{S}\}$ and $\textsc{pred}(x, \mathcal{S})$ is defined symmetrically.

We further introduce the first cache-oblivious hash table, the **Cache-Oblivious Bundle of Trees Hash Table** (COBOT), which matches the IO performance of BOTs and IP hash tables.

The BOT can also be adapted to models in which disk reads and writes incur different costs. The β-asymmetric BOT adjusts the merging schedule of a regular BOT to trade some writes for more reads. We leave the details for the full version [**?**].

▶ **Theorem 4.** *A β-asymmetric BOT supports N insertions and deletions with amortized per entry cost of $O\left(\frac{1}{B}\left(\lambda + \frac{1}{\beta}\log_\lambda N\right)\right)$ writes and $O\left(\frac{1}{B}\left(\lambda+\beta\right)\right)$ reads for any $\lambda > 1$ and $\beta \leq \left\lfloor \log_\lambda \frac{M}{B\log_\lambda N}\right\rfloor$. A query for a key K performs $O(D_K \log_\lambda N)$ reads, where D_K is the number of times K has been inserted or deleted.*

2 Preliminaries

Fingerprints and Hashing. In order to achieve our bounds, we need $\Theta(\log N)$-wise independent hash functions, which, once again matches IP hash tables. We note that a k-wise independent hash function is also k-wise independent on individual bits. Furthermore, the following Chernoff-type bound holds:

▶ **Lemma 5** ([14]). *Let $X_1, X_2, \ldots, X_N$ be $\lceil \mu\delta \rceil$-wise independent binary random variables, $X = \sum_{i=1}^{N} X_i$ and $\mu = \mathrm{E}\left[X\right]$. Then $\mathrm{P}\left(X > \mu\delta\right) = O\left(1/\delta^{\mu\delta}\right)$, for sufficiently large δ.*

In the following, we use **fingerprint** to refer to any key that has been hashed using a $\Theta(\log N)$-wise independent hash function. Such hash functions have a compact representation and can be specified using $\Theta(\log N)$ words. The universe that is hashed into is assumed to have size $\Theta(N^k)$ for $k \geq 2$. We ignore collisions, but these can be handled as in [12].

For a fingerprint K, it will be convenient to interpret the bits of K as a string of $\log \lambda$ (where lambda is a given parameter) bit characters, $K = K_0K_1K_2\cdots$.

Delta Encoding. We will frequently encounter sorted lists of fingerprint prefixes (possibly with duplicates), together with some data about each. When the size of the list is dense in the space of prefixes, we can compress it using **delta encoding**, where the difference between prefixes is stored rather than the prefixes themselves.

▶ **Lemma 6.** *A list of delta-encoded prefixes with density D, that is there are D prefixes in the list for every possible prefix, requires $O(-\log_\lambda D)$ characters per prefix.*

Proof. The average difference between consecutive prefixes is $1/D$. Because logarithms are convex, the average number of characters required to represent this difference is therefore $O(-\log_\lambda D)$. ◀

Log-structured Merge Trees. Log-structured merge trees (LSMs) are (a family of) external-memory dictionary data structures. They come in two varieties: **level-tiered LSMs** (LT-LSMs) and **size-tiered LSMs** (ST-LSMs). Both kinds are suboptimal in that they do not meet the optimal insertion-query tradeoff [8], although the COLA [3] is an optimal variant of the LT-LSMs.

An LSM consists of sets of either B-trees or sorted arrays called **runs**. In this paper, we describe them in terms of runs, since we use runs below.

An LT-LSM consists of a cascade of levels, where each level consists of at most one run. Each level has a **capacity** that is λ times greater than the level below it, where λ is called the **growth factor**.[2] When a level reaches capacity, it is merged into the next level (perhaps causing a merge cascade). The amortized IO cost for insertions is small because sequential merging is fast, although each item will participate in $\lambda/2$ merges on average. The IO cost for a query is high because a query must be performed independently on each of $O(\log_\lambda N)$ levels (although Bloom filters [5, 7] are sometimes used to mitigate this cost).

An ST-LSM further improves insertion IOs at the expense of queries. Each level contains fewer than λ runs. Every run on a given level has the same size, which is λ times larger than the runs on the level beneath it. When λ runs are present at a level, they are merged into one run and placed at the next level. There are therefore $O(\log_\lambda N)$ levels. Insertions are faster than in LT-LSMs because each item is only merged once on each level. Queries are slower because each query must be perform $O(\lambda)$ times at each level.

In LSMs, deletions can be implemented by the use of **upsert messages** [9, 13], which are a type of insertion with a message that indicates that the key has been deleted. A query for a key K then fetches all the matching key-value pairs and if the last one (temporally) is a deletion upsert, it returns false. To this end, the merges must maintain the temporal order of key-value pairs with the same key. Because a query for a key K must fetch every instance of K, the cost of a query is proportional to the number of times the key has been inserted and deleted, which we refer to as the **duplication count**, D_K of K. When $N/2$ deletions have been made, the structure is rebuilt to reclaim space. In what follows, deletions will be implemented using the same mechanism.

3 Bundle of Arrays Hashing

A **Bundle of Arrays Hash Table** (BOA) is an external-memory dictionary based on ST-LSMs. In this section, we describe a simple version which is optimal in the sense of Theorem 1, but where the query cost meets the bound only in expectation, not w.h.p. In Section 4, we give a version that satifies Theorem 2.

As a first step, we show that runs with uniformly distributed, $\Theta(\log N)$-wise independent fingerprints can be searched more quickly than in an ST-LSMs. In this section

▶ **Lemma 7.** *Let A be a sorted array of N uniformly distributed $\Theta(\log N)$-wise independent keys in the range $[0, K)$, and assume $B = \Omega(\log N)$. Then A can be written to external memory using $O(N)$ space and $O(N/B)$ IOs so that membership in A can be determined in $O(1)$ IOs with high probability.*

Proof. First note that, by Lemma 5 and Bonferroni's inequality, if N balls are thrown into $\Theta(N/\log N)$ bins uniformly and $\Theta(\log N)$-wise independently, then every bin has $\Theta(\log N)$ balls with high probability.

Divide the range of keys into N/B uniformly sized buckets; that is, bucket i contains keys in the range $[(i-1)KB/N, iKB/N)$. Because the keys in A are distributed uniformly, and $B = \Omega(\log N)$, every bucket contains $\Theta(B)$ keys with high probability. Let F be the number of items in the fullest bucket, and write the keys in each bucket to disk in order using F space for each. Because $F = \Theta(B)$, this takes the desired space and IOs.

Now, to find a key, compute which bucket it belongs to. A constant number of IOs will fetch that bucket, whose address is known because all buckets have the same size. ◀

[2] Sometimes this and related structures are analyzed with a growth factor of B^ϵ. The two are equivalent. We use λ rather than ϵ as the tuning parameter for consistency with the external-memory hashing literature.

▶ **Corollary 8.** *If an ST-LSM contains uniformly distributed and $\Theta(\log N)$-wise independent fingerprints and has growth factor λ, then a query for K can be performed in $O(D_K \lambda \log_\lambda N)$ IOs by writing the levels as in Lemma 7. The insertion/deletion cost is unchanged: $O\left(\frac{1}{B}\left(\log_\lambda N + \log_{\frac{M}{B}} N\right)\right)$ amortized IOs.*

While the query performance improves by a factor of $\log N$, the ST-LSM is still off the optimal tradeoff curve of Theorem 1. In particular, queries can be at least exponentially slower than optimal. The BOA use additional structure in order to reduce this query cost.

3.1 Routing Filters

The main result of this section is an auxiliary data structure, the **routing filter**, that improves the query cost of an ST-LSM by a factor of λ by further exploiting the log-wise independence of fingerprints. Combining these routing filters with fast interpolation search will yield the BOA, a hashing data structure that is optimal for large enough λ.

The purpose of the routing filter is to indicate probabilistically, at each level, which run contains the fingerprint we are looking for. Each level will have its own routing filter, defined as follows. For each level ℓ, let h_ℓ be some number, to be specified below. Let $P_\ell(K)$ be the prefix consisting of the first h_ℓ characters of K. The routing filter F_ℓ for level ℓ is a λ^{h_ℓ}-character array, where $F_\ell[i] = j$ if the jth run $R_{\ell,j}$ contains a fingerprint K such that the $P_\ell(K) = i$, and no later run $R_{\ell,j'}$ (i.e. with $j' > j$) contains such a fingerprint.

We also modify each run $R_{\ell,j}$ during the merge so that each fingerprint-value pair contains a **previous field** of 1 additional character used to specify the previous run containing a fingerprint with the same prefix, or j to indicate no such run exists. Thus these fingerprint-value pairs now form a singly linked list whose fingerprint share the same prefix, and the routing filter points to the run containing the head.

During a query for a fingerprint K, first $F_\ell[P_\ell(K)]$ is checked to find the latest run containing a fingerprint with a matching prefix. Once that fingerprint-value pair is found, its previous field indicates the next run which needs to be checked and so on until all fingerprints with matching prefix in the level are found. Each fingerprint $K' \neq K$ that matches K's prefix is a false positive.

Such routing filters induce a space/cost tradeoff. The greater h_ℓ is, the more space the table takes but the less likely it is that many runs will have false positives. The rest of this section shows that when $h_\ell = \log_\lambda B + \ell$, in other words, when prefixes grow by a character per level, the BOA lies on the optimal tradeoff curve of Theorem 1.

Define β, the **routing table ratio**, to be the ratio of the number of buckets in the routing filter to the size of a run. The number of entries in a run on level ℓ is $B\lambda^{\ell-1}$, so $\beta = \lambda^{h_\ell}/B\lambda^{\ell-1}$. We first analyze the per-level insertion/deletion cost, and then we compute the expected number of false positives in order to analyze the overall query cost.

▶ **Lemma 9.** *For a BOA with growth factor λ and routing table ratio β, merging a level incurs $\Theta\left(\frac{1}{B}\left(1 + \log_{\frac{M}{B}} \lambda + \beta \log_N \lambda\right)\right)$ IOs per fingerprint.*

Proof. Merging a level requires merging its runs as well as updating the next level's routing filter. Merging λ sorted arrays takes $\Theta\left(\frac{1}{B}\left(1 + \log_{M/B} \lambda\right)\right)$ IOs per fingerprint.

The routing filter is updated by iterating through it and the new run sequentially. For each fingerprint K appearing in the run, $F_{\ell+1}[P_{h_{\ell+1}}(K)]$ is copied to the previous field in the run, and $F_{\ell+1}[P_{h_{\ell+1}}(K)]$ is set to the number of the current run. Each entry in the routing filter is a character, and the routing filter has β entries for each new fingerprint. Thus, it requires $\Theta\left(\frac{\beta}{B} \log_N \lambda\right)$ IOs per fingerprint to update sequentially. ◀

▶ **Lemma 10.** *For a BOA with growth factor λ and routing table ratio β, querying a fingerprint K on a given level incurs at most $\frac{\lambda}{\beta}$ false positives in expectation.*

Proof. Given some enumeration of the fingerprints in level ℓ, which are not equal to K, denote the ith such fingerprint by K_i. Some of these may be the duplicates. Let X_i be the indicator random variable, which is 1 if $P_\ell(K) = P_\ell(K_i)$ and 0 otherwise. K and K_i are uniformly distributed and their bits are pairwise independent. Thus $\mathrm{E}[X_i] \le \frac{1}{\lambda^{h_\ell}}$. The expected number of fingerprints (excluding K) in the level with prefix $P_\ell(K)$ is thus at most $\sum_{i=1}^{B\lambda^\ell} \mathrm{E}[X_i] \le \frac{B\lambda^\ell}{\lambda^{h_\ell}} = \frac{\lambda}{\beta}$. ◀

▶ **Lemma 11.** *A BOA with growth factor λ and routing table ratio β has insertion/deletion cost $O\left(\frac{1}{B}\left(\beta + \log_{\frac{M}{B}} N + \log_\lambda N\right)\right)$. A query for fingerprint K has expected cost $O\left(\frac{\lambda}{\beta} D_K \log_\lambda N\right)$, where D_K is the duplication count of K.*

Proof. Because a BOA has $\log_\lambda N$ levels, the insertion cost follows from Lemma 9.

To query for a fingerprint K, the routing filter on each level is checked, which incurs $O(\log_\lambda N)$ IOs. These routing filters return a collection of runs which contain up to D_K true positives and an expected $O\left(\frac{\lambda}{\beta}\log_\lambda N\right)$ false positives, by Lemma 10. By Lemma 7, each run can be checked in $O(1)$ IOs. ◀

So for a fixed λ, there is no advantage to choosing $\beta = \omega(\lambda)$. On the other hand, $\beta = o(\lambda)$ is suboptimal, because then choosing $\beta' = \lambda' = \beta$ changes a linear factor in the query cost to a logarithmic one. Therefore, it is optimal to choose $\beta = \Theta(\lambda)$, and in what follows we will fix $\beta = \lambda$. Thus,

▶ **Lemma 12.** *A BOA supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log_\lambda N\right)/B\right)$ IOs, for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs in expectation, where D_K is the duplication count of K.*

4 Refined Bundle of Arrays Hashing

In order to obtain high probability bounds for a BOA, we need a stronger guarantee on the number of false positives. This is achieved by including an additional character, the **check character** from each fingerprint in the routing filter, which is also checked during queries and thus eliminates most false positives. To support this, we will need to refine the routing filter so it can maintain check characters even when there are collisions.

The ith check character $C_i(K)$ of a fingerprint K is the ith character from the end of the string representation of K. As described in Section 2, we assume that the fingerprints are taken from a universe of size at least N^2 so that the check characters do not overlap with the characters used in the prefixes of the routing filters, and by $\Theta(\log N)$-wise independence, the check characters of $O(1)$ fingerprints are independent. Now each fingerprint in the filter has a check character and a array pointer, and we refer to this data as the **sketch** of the fingerprint.

When level i of the BOA is queried for a fingerprint K, the refined routing filter (described below) returns a list of sketches, one for each fingerprint in the level with prefix $P_i(K)$. The array indicated in the sketch is only checked if the check character matches $C_i(K)$, which reduces the number of false positives by a factor of λ.

Refined Routing Filter. The routing filter described in Section 3.1 handles prefix collisions by returning only the last run containing the queried fingerprint and then chaining in the runs. Whereas, to support check characters, we need to return a list instead, while having the same performance guarantees.

The idea behind the refined routing filter is to keep the prefix-sketch pairs in a sorted list and use a hash table on prefixes to point queries to the appropriate place. Each pointer may require as many as $\Omega(\log N)$ bits, and we require the routing filter to have $O(1)$ characters per fingerprint. Therefore the hash table must use shorter prefixes so as to reduce the number of buckets and thus reduce its footprint. In particular, it uses prefixes which are $\log_\lambda \log_\lambda N$ characters shorter, which we refer to as **pivot prefixes**.

The list delta encodes the prefix for each fingerprint K, together with its sketch. In addition, the first entry following each pivot prefix contains the full prefix, rather then just the difference. Otherwise, when the hash table routes a query to that place in the list, the full prefix wouldn't be immediately computable.

▶ **Lemma 13.** *A refined routing filter can be updated using $O\left(\frac{\lambda \log \lambda}{B \log N}\right)$ IOs per new entry, and performs lookups in $O(D_K^*)$ IOs w.h.p., where D_K^* is the number of times K appears in the level.*

Proof. We prove first the update bound and then the query bound.

Let C be the capacity of the level. There are at most $\frac{C}{\log_\lambda N}$ pivot prefixes. For each pivot prefix, the hash table stores the bit position in a list with at most C entries, where $C \leq N$. Each entry is at most $\log N$ bits, so this position can be written using $O(\log N)$ bits.

For each fingerprint in the node, the list contains $O(1)$ characters by Lemma 6, or $O(\log \lambda)$ bits. Additionally, each pivot prefix has to an initial entry of length $O(\log N)$ bits, so the list all together uses $O(C \log \lambda + \frac{C}{\log_\lambda N} \cdot \log N) = O(C \log \lambda)$ bits.

When the refined routing filter is updated, the old version is read sequentially and the new version is written out sequentially. C/λ fingerprints are added at a time, so this incurs $O\left(\frac{\lambda \log \lambda}{B \log N}\right)$ IOs per entry.

During a query, the pivot bit string of a fingerprint and its successor are accessed from the hash table in $O(1)$ IOs. This returns the beginning and ending bit positions in the list. Because the fingerprints are distributed uniformly and are pairwise independent to K, there are $O(\log_\lambda N + D_K^*)$ fingerprints matching the pivot prefix in expectation. From Lemma 5 with $\delta = \log \lambda$, there are $O(\log N + D_K^*)$ fingerprints matching the pivot prefix w.h.p. The encoding of each fingerprint is less than a word, and $B = \Omega(\log N)$ by assumption, so this is $O(D_K^*)$ IOs. ◀

BOA Performance. We now can show:

▶ **Theorem 2.** *A BOA supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log_\lambda N\right)/B\right)$ IOs, for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

Proof. The insertion/deletion cost is given by Lemma 13 and Theorem 12.

During a query for a fingerprint K, the expected number of false positives on level i (fingerprints which match the prefix $P_i(K)$ and check character $C_i(K)$ but are not K) is $O\left(\frac{1}{\lambda}\right)$. Thus, the number of false positives across levels is $O\left(\frac{\log_\lambda N}{\lambda}\right)$, so by Lemma 5, the number of false positives is $O\left(\log_\lambda N\right)$ w.h.p. ◀

Thus, a BOA is optimal for large enough λ:

▶ **Corollary 14.** *Let $\mathcal{B}$ be a BOA with growth factor λ containing N entries. If $\lambda = \Omega\left(\log_{\frac{M}{B}} N + \frac{\log N}{\log \log N}\right)$, then $\mathcal{B}$ is an optimal unsorted dictionary.*

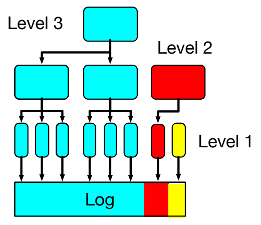

Figure 1 The routing trees in a 3 level BOT. The trees cover contiguous portions of the log. The highest level covers the beginning of the log, the next level the beginning of the remainder of the log, and so on.

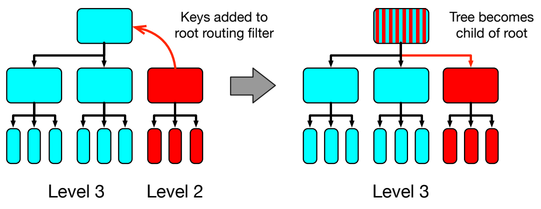

Figure 2 When the routing tree on level i fills, it is merged into the routing tree on level $i+1$. The now-full routing tree from level $i+1$ becomes a child of the root on level $i+1$. Its fingerprints are added to the root routing filter. Note that the tree is not moved.

5 Bundle of Trees Hashing

In order for a BOA to be an optimal dictionary, its growth factor λ must be $\Omega(\log N/\log\log N)$. Otherwise, the cost of insertion is dominated by the cost of merging, which in slow because it effectively sorts the fingerprints using a λ-ary merge sort. In this section, we present the **Bundle of Trees Hash Table** (BOT), which is a BOA-like structure. A BOT stores the fingerprints in a log in the order in which they arrive. Each level of the BOT is like a level of a BOA, where the bundle of arrays on each level is replaced by an search structure on the log (the **routing tree**) and a data structure needed to merge routing trees (the **character queue**). The character queue performs a delayed sort on the characters needed at each level, thus increasing the arity of the sort and decreasing the IOs.

A BOT has $s = \lceil \log_\lambda N/B \rceil$ levels, each of which contains a routing tree. The root of the routing tree has degree less than λ and all internal nodes have degree λ. Each node of a routing tree contains a routing filter. As in Section 4, each routing filter takes as input a fingerprint K and outputs a list of sketches corresponding to fingerprints with the same prefix as K. Each sketch consists of a pointer to a child, a check character and some auxiliary information discussed below.

Each leaf points to a block of B fingerprints in the log. The deepest level s uses a height-s tree to index the beginning of the fingerprint log, the next level then indexes the next section and so forth, as shown in Figure 1. Insertions and deletions (as upsert messages) are appended to the log until they form a block, at which point they are added to the tree in the 1st level of the BOT.

When a level i in the BOT fills, its routing tree is merged into the routing tree of level $i+1$, thus increasing the degree of the target routing tree by 1 (and perhaps filling level $i+1$, which triggers a merge of level $i+1$ into $i+2$, and so on). The merge of level i into level $i+1$ consists of adding the prefix-sketch pairs of the fingerprints from level i to the routing filter of the root on level $i+1$. The child pointers of these pairs will point to the root of the formerly level-i routing tree, so it becomes a child of the root of the level $i+1$ routing tree, although it isn't moved or copied. See Figure 2. In this way, a BOT resembles an LT-LSM, described in Section 2.

In order to add a fingerprint K from level i to the root routing filter on level $i+1$, the prefix $P_{i+1}(K)$ must be known. However, the root routing filter on level i only stores the prefix $P_i(K)$ for each fingerprint K it contains, so that in particular the last character of $P_{i+1}(K)$ is missing. As described in Section 5.2, each level has a character queue, which provides this character, as well as the check characters, in order to merge the routing trees efficiently.

5.1 Queries in a BOT

A query to the BOT for a fingerprint K is performed independently at each level, beginning at the root of each routing tree. When a node is queried, its routing filter returns a list of sketches. The sketches whose check characters match the queried fingerprint indicate to which children the query is passed. This process continues until the query reaches a block of the log, which is then searched in full. In this way queries are "routed" down the tree on each level to the part of log where the fingerprint and its associated value are. Note that as queries descend the routing tree, they may generate false positives which are likewise routed down towards the log.

In this section, we refine routing trees so that they offer two guarantees about false positives. The first is that at each level, the probability that a given false positive is not elinimated is at most $\frac{1}{\lambda}$. The second is that false positives can only be created in the root, so that as the query descends the tree, the number of false positives cannot increase.

During a query to a node of height h for a fingerprint K, the routing filter returns a list of sketches corresponding to fingerprints which match K's prefix. The query only proceeds on those children whose check characters also match the check character $C_h(K)$. Since the characters of the fingerprint are uniformly distributed and $\Theta(\log N)$-wise independent, the check character of each false positive matches with probability $\frac{1}{\lambda}$. Moreover, the characters of each level are non-overlapping, so for fingerprints K, K' the event that $V_h(K) = V_h(K')$ is independent of the event that $V_{h-1}(K) = V_{h-1}(K')$.

To prevent new false positives from being generated when a query passes from a parent to a child, the **next character** of each fingerprint is also kept in its sketch in the routing filter. For a fingerprint K in a node of height h, the next character is just the next character that follows the prefix, $P_h(K)$, so that its prefix in the parent, $P_{h+1}(K)$, can be obtained. A false positive in a child which is not in the parent will not match this next character and can be eliminated.

When there are multiple prefix-matching fingerprints in both a parent and its child, we would like to be able to align the lists returned by the routing filters so that known false positives in the parent (either from check or next characters) can be eliminated in the child. Otherwise the check character in the child of a known false positive in the parent may match the queried fingerprint, and therefore more than $\frac{1}{\lambda}$ of the false positives may survive. To this end, we require the routing filter to return the list of sketches in the order their fingerprint-value pairs appear in the log. Then after the sketches in the child list whose next characters do not match the parent are eliminated, the remaining phrases will be in the same order as in the parent. In this way, known false positives can also be eliminated in the child.

Now we can show:

▶ **Lemma 15.** *During a query to a routing tree, the following are true:*

1. *A false positive can only be generated in the root.*
2. *At each level, a given false positive survives with probability at most $\frac{1}{\lambda}$.*

Proof. Because of the next characters, false positives may only be created in the root of the routing tree. Each false positive in the root corresponds to a fingerprint K' in the level. At each node on the path to K''s location in the log, we use the ordering to determine which returned sketch corresponds to K', so that the false positive corresponding to K' is eliminated with probability $\frac{1}{\lambda}$. ◀

5.2 Character Queue

The purpose of the character queue is to store all the sketches of fingerprints contained in a level i that will be needed during a merge in the future. When level i is merged into level $i+1$, the character queue outputs a sorted list of the delta-encoded prefix-sketch pairs of all the fingerprints, which is used to update the root routing tree. The character queue is then merged into the character queue on level $i+1$.

The character queue effectively performs a merge sort on the sketches. If it were to merge all the sketches as soon as they are available, this would consist of λ-ary merges. In order to increase the arity of the merges, it defers merging sketches which are not needed immediately. The sketches are stored collection of **series**, by which we mean a collection of sorted runs. Each series stores a continuous range of sketches $S_i(K), S_{i+1}(K), \ldots, S_{i+j}(K)$ for each fingerprint K, together with the prefix up to the first sketch, $P_{i-1}(K)$. These prefixes are delta encoded in their run. Thus the size of an entry is determined by the number of sketches in the range and the length of the prefix relative to the size of the run (by Lemma 6).

The character queue tradeoff. We are faced with the following tradeoff. If the character queue merges a series frequently, the delta encoding is more efficient, which decreases the cost of the merging. However the arity is lower, which increases it. The character queue uses a merging schedule which balances this tradeoff and thus achieves optimal insertions.

The character queue merging schedule. The character queue on level i (here we consider blocks of the log to be level 0) contains the sketches $S_{i+1}(K), S_{i+2}(K), \ldots S_s(K)$ of each fingerprint K in the level. These characters are stored in a collection of series $\{\sigma_{j_q}\}$, where j_q is the smallest multiple of 2^q greater than i. Series σ_{j_q} contains the sketches $S_{j_q}(K), \ldots, S_{j_{q+1}-1}(K)$. Each series consists of a collection of sorted runs each of which stores the delta encoded prefix of each fingerprint together with its sketches.

Initially, when a block of the log is written, all the series σ_{2^q} for $q = 1, 2, 3, \ldots$ are created. When level i fills, the runs in the series σ_{i+1} are merged, and the character queue outputs the delta encoded prefix-sketch pairs, $(P_{i+1}(K), S_{i+1}(K))$ to update the root routing filter on level $i+1$. If $2^{\rho(i+q)}$ is the greatest power of 2 dividing $i+1$ (ρ is sometimes referred to as the **ruler function** [15]), then σ_{i+1} also contains the next $2^{\rho(i+1)} - 1$ sketches of each fingerprint. These are batched and delta encoded to become runs in the series σ_{j_q} for $q = [0, \rho(i+1)]$. The runs in the remaining series of level i becomes runs of their respective series on level $i+1$.

Note that for the lower levels, some runs in may be shorter than B due to the delta encoding. For a run in a series σ_q, this is handled by buffering them with the runs σ_q of higher levels and writing them out once they are of size B. Note that this requires $O(B \log\log N)$ memory.

This leads to the following merging pattern: σ_j batches $2^{\rho(j)}$ sketches, and has delta encoded prefixes of $2^{\rho(j)}$ characters on average, by Lemma 6. Therefore,

▶ **Lemma 16.** *A series σ_j in a character queue contains $O(2^{\rho(j)})$ characters per fingerprint.*

This leads to a merging schedule where the characters per item merged on the jth level is $O(2^{\rho(j)})$. Starting from 1 this is $1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2, 1, 16, \ldots$, which resemble the tick marks of a ruler, hence the name ruler function.

We now analyze the cost of maintaining the character queues.

▶ **Lemma 17.** *The total per-insertion/deletion cost to update the character queues in a BOT is $\Theta\left(\frac{1}{B}\left(\log_{\frac{M}{B}} N + \log\log M\right)\right)$.*

Proof. When σ_j is merged, $\lambda^{2^{\rho(j)}}$ runs are merged, which has a cost of $O\left(\frac{2^{\rho(j)}}{B}\left\lceil\log_{M/B}\left(\lambda^{2^{\rho(j)}}\right)\right\rceil\right)$ characters per fingerprint.

There are $\log_\lambda \frac{N}{B} = O(\log_\lambda N)$ levels, so this leads to the following total cost in terms of characters:

$$\begin{aligned} O\left(\sum_{i=1}^{\log_\lambda N} 2^{\rho(j)}\left\lceil\log_{\frac{M}{B}}\left(\lambda^{2^{\rho(j)}}\right)\right\rceil\right) &= O\left(\sum_{k=0}^{\log\log_\lambda N} \frac{\log_\lambda N}{2^k}\cdot 2^k\left\lceil\log_{\frac{M}{B}}\left(\lambda^{2^k}\right)\right\rceil\right) \\ &= O\left(\log_\lambda N\left(\log\log M + \sum_{k=\log\log M}^{\log\log_\lambda N} 2^k \log_{\frac{M}{B}}\lambda\right)\right) \\ &= O\left(\log_\lambda N\left(\log\log M + \log_{\frac{M}{B}} N\right)\right), \end{aligned}$$

where the last equality is because the RHS sum is dominated by its last term. Because there are $\log_\lambda N$ characters in a word, and all reads and writes are performed sequentially in runs of size at lease B, the result follows. ◀

5.3 Performance of the BOT

We can now prove Theorem 3:

▶ **Theorem 3.** *A BOT supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log\log M\right)/B\right)$ IOs for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

Proof. By Lemma 13, the cost of updating the routing filters is $O\left(\frac{\lambda}{B}\right)$, since there are $O(\log_\lambda N)$ levels. This, together with the cost of updating the character queues, given by Lemma 17, is the insertion cost.

By Lemma 15, a query for fingerprint K on level i incurs $O\left(\frac{1}{\lambda}\right)$ false positives in the root, and $O(1)$ nodes are accessed along each of their root-to-leaf paths. By Lemma 13, each false positive thus incurs $O(D_K)$ IOs.

There are an expected $O\left(\frac{\log_\lambda N}{\lambda}\right)$ false positives across all levels, so, using Lemma 5 with $\delta = \lambda$, $O(D_K \log_\lambda N)$ nodes are accessed due to false positives w.h.p. For each time K appears in the BOT, $O(\log_\lambda N)$ nodes are accessed on its root-to-leaf path. By Lemma 13 the node accesses along each path incur $O(D_K \log_\lambda)$ IOs w.h.p., so accessing the nodes incurs $O(\log_\lambda N)$ IOs w.h.p.

A block of the log is scanned at most D_K times for true positives and also whenever a false positive from the level-i root survives i times. The expected number of such false positives for level i is $1/\lambda^i$, so the expected number across levels is $O\left(\frac{1}{\lambda}\right)$. Therefore by Lemma 5, the number of blocks scanned is $O(D_K \log_\lambda N)$ w.h.p. ◀

▶ **Corollary 18.** *Let $\mathcal{B}$ be a BOT with growth factor λ containing N entries. If $\lambda = \Omega\left(\log_{\frac{M}{B}} N + \log\log M\right)$, then $\mathcal{B}$ is an optimal dictionary.*

6 Cache-Oblivious BOTs

In this section, we show how to modify a BOT to be cache oblivious. We call the resulting structure a cache-oblivious hash tree (COBOT).

Much of the structure of the BOT translates directly into the cache-oblivious model. However, some changes are necessary. In particular, when the series of character queues are merged, this merge must be performed cache-obliviously using funnels [10], rather than with an (up to) M/B-way merge. Also, the log cannot be buffered into sections of size $O(B)$, and so instead they are buffered into sections of constant size, items are immediately added to routing filter, and the extra IOs are eliminated by optimal caching.

When an insertion is made into a COBOT, its fingerprint-value pair is appended to the log, and it is immediately inserted into level 1. Thus, the leaves of the routing trees point to single entries in the log.

The series of the character queues must be placed more carefully as well. In particular the runs of series σ_j must be laid out back-to-back for all j (rather than just small j as in Section 5.2), so that the caching algorithm can buffer them appropriately.

The series are merged using a **partial funnelsort**. Funnelsort is a cache-oblivious sorting algorithm that makes use of K-funnels [10]. A K-funnel is a CO data structure that merges K sorted lists of total length N. We make use of the the following lemma.

▶ **Lemma 19** ([10]). *A K-funnel merges K sorted lists of total length $N \geq K^3$ in $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + K + \frac{N}{B}\log_K\frac{N}{B}\right)$ IOs, provided the tall cache assumption that $M = \Omega(B^2)$ holds.*

The partial funnelsort used to merge K runs of a series with total length L (in words) performs a single merge with a K-funnel if $L \geq K^3$ and recursively merges the run in groups of $K^{1/3}$ runs otherwise.

▶ **Corollary 20.** *A partial funnelsort merges K runs of total word length L in $O\left(\frac{L}{B}\log_{M/B}\frac{L}{B} + \frac{L}{B}\log_K\frac{L}{B}\right)$ IOs, provided the tall cache assumption that $M = \Omega(B^2)$ holds.*

Proof. The base case of the recursion occurs either when there is only 1 list remaining or the remaining lists fit in memory. In any other case of the recursion, since $L = \Omega(B^2)$ by the tall cache assumption, the K term in Lemma 19 is dominated.

The recurrence is dominated by the cost of the funnel merges, which yields the result. ◀

▶ **Theorem 21.** *If $M = \Omega(B^2)$, then a COBOT with N entries and growth factor λ has amortized insertion/deletion cost $\Theta\left(\frac{1}{B}\left(\lambda + \log\log M + \log_{M/B} N/B\right)\right)$. A query for key K has cost $\Theta\left(D_K \log_\lambda N\right)$, w.h.p., where D_K is the duplication count of K.*

Proof. We may assume that the caching algorithm sets aside enough memory that the last B items in the log, together with the subtree rooted at their least common ancestor, are cached. Thus the log is updated at a per-item cost of $O(1/B)$.

The proof of Theorem 3 now carries over to the COBOT. The routing filters are updated the same way, and the cost of updating the character queues is unchanged, by Corollary 20.

Queries are performed as in Section 5.1, except that now the level 1 nodes cover $O(1)$ fingerprints, but the depth of the tree is unchanged, so the cost is the same. ◀

References

1 Peyman Afshani, Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Mayank Goswami, and Meng-Tsung Tsai. Cross-referenced dictionaries and the limits of write optimization. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1523–1532. SIAM, 2017. doi:10.1137/1.9781611974782.99.

2 Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151. Springer, 2002. `doi:10.1007/3-540-45749-6_16`.

3 Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 81–92. ACM, 2007. `doi:10.1145/1248377.1248393`.

4 Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Meng-Tsung Tsai. The batched predecessor problem in external memory. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 112–124. Springer, 2014. `doi:10.1007/978-3-662-44777-2_10`.

5 Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012. URL: `http://vldb.org/pvldb/vol5/p1627_michaelabender_vldb2012.pdf`.

6 Michael A. Bender, Martin Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. Write-optimized skip lists. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 69–78. ACM, 2017. URL: `http://dl.acm.org/citation.cfm?id=3034786`, `doi:10.1145/3034786.3056117`.

7 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. `doi:10.1145/362686.362692`.

8 Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 546–554. ACM/SIAM, 2003. URL: `http://dl.acm.org/citation.cfm?id=644108.644201`.

9 John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The tokufs streaming file system. In *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012*, 2012. URL: `https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/esmet`.

10 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6604`, `doi:10.1109/SFFCS.1999.814600`.

11 John Iacono and Mihai Patrascu. Using hashing to solve the dictionary problem (in external memory). *CoRR*, abs/1104.2799, 2011. `arXiv:1104.2799`.

12 John Iacono and Mihai Patrascu. Using hashing to solve the dictionary problem. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 570–582. SIAM, 2012. URL: `http://portal.acm.org/citation.cfm?id=2095164&amp;CFID=63838676&amp;CFTOKEN=79617016`, `doi:10.1137/1.9781611973099`.

13 William Jannen, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Lazy analytics: Let other queries do the work for you. In

8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, June 20-21, 2016., 2016. URL: https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/jannen.

14 Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995. doi:10.1137/S089548019223872X.

15 Wikipedia. Thomae's function — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Thomae's%20function&oldid=837510765, 2018. [Online; accessed 28-April-2018].