Optimal Hashing in External Memory [1]

Mario Stöckli
27.10.18

1 Disclaimer

This report refers to the paper Optimal Hashing in External Memory, written by Alex Conway, Martín Farach-Colton and Philip Shilane and further sources cited in that paper. [1].

2 Introduction

The paper is about building a dictionary residing in external memory with the best possible bounds for this problem. Even though an optimal solution already has been found with the IP hash table, the authors have found a different, simpler version with a small disadvantage, that the lookup time of the key worsens everytime the key gets updated. The reason why we would want to have a dictionary in external memory is simple. We might want to have a dictionary so big, that we would have to spend an absurd amount on money on fast, expensive local memory to have enough space. We therefore prefer cheaper, but slower external memory for huge datasets. But before we get into the details of the data structure the authors propose, we will look shortly at the problem itself at hand and the current state of the art to understand these bounds and the process of arriving at such a data structure. Even though the authors start with a simple non-optimal idea, the Bundle of Arrays Hash Table (BOA), and iterate towards their optimal solution, the COBOT (Cache-Oblivious Bundle of Trees Hash Table), we need to take a step further back to get a clearer picture of the situation.

3 Definitions

Let B be the block size, M the cache size, N the number of objects to be within the dictionary and λ be a constant tuning parameter, which is determined by the "growth factor" of the log-structured merge tree (LSM). We will see how λ comes into play within the Bundle of Arrays Hash Table (BOA). Fingerprint: A key, hashed by a $\theta(\log N)$-wise independent hash function. For a fingerprint K, we will interpret it as a string of λ bit characters: $K = K_0K_1K_2...$

3.0.1 Abbreviations

w.h.p.: with high probability
I/O: Input/Output operation; a memory transfer between local and external memory

4 The problem

When we analyze the runtime of algorithms, we usually count the number of elementary operations and do an asymptotic or amortized analysis according to the input size N. In the paper there are two important things to remark: The paper is based on the DAM (disk-access model) and the cache-oblivious model. These models have in common, that they have the data transfer between local memory and external memory as their elementary operation, instead of the computational effort

within the CPU. We do this, as I/O is much slower than most calculations within the CPU, so we can do these "for free" while waiting for the data from the external memory (or while waiting until it's written). So as long as calculations within memory are not a thing yet, we have to accept the I/Os as dominating factor that determines our bounds.

This could result in non-optimal solutions, especially if we consider our case, where we want to store a dictionary in external memory. We never know in advance in what order insertions, deletions and queries will appear. This could result in non-optimal behaviour, because we want to use the full capacity of B for every I/O, to achieve the lowest possible amount of I/Os and we can only do this as we access whole blocks of size B to read or write, so we have to place the data in a smart way.

So we also realize we have a non-trivial problem worth solving, if we imagine a naive external dictionary (hash table). If we update everytime for a single key-value pair, we have for N updates runtime $O(N)$ instead of the optimal $O(N/B)$ within the DAM. Then we could say, we simply store in order of arrival within blocks in $O(N/B)$. But this would destroy our query performance, because we don't store according to a hashing function anymore and therefore can't find our element in $O(1)$.

In the following section I'm going to explain these concepts and their backgrounds.

4.1 The disk-access model [2]

The DAM, first introduced by Aggarwal and Vitter in 1988, is a simplification of the modern architectures. It consists of a CPU, directly connected to a cache with capacity M. Connected to that cache is the disk, which has unbounded capacity and can be divided into blocks of size B. The cache can therefore hold $\lfloor M/B \rfloor$ Blocks. This makes sense, as B corresponds to the size of one memory transfer in the model, which we associate with a cost of 1. So if we have for example a program, which has to load $X \leq M$ data into the cache, we need $\lceil X/B \rceil$ memory transfers, what would make our program finish in $O(X/B)$. This model is close to reality, because a disk can profit from sequential access. Furthermore if we compare it to RAM, which most modern computers have, we could transfer an algorithm from a RAM model to this model: We simply use RAM as our bus and load anything from disk to RAM and then from there to Cache, thus ignoring RAM, which makes I/Os only slower by a constant factor. Note that in the DAM model more than just comparisons are allowed, which are free, as we only count I/Os. An illustration can be seen in figure 1 below.

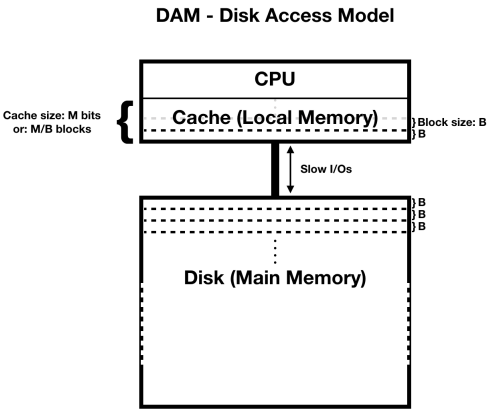

Figure 1: DAM model

4.2 The cache oblivious model [3]

The main idea of the cache oblivious model is to construct an algorithm, which works optimally even for unknown B and M. This would be great to have, as we don't want to have to change our algorithm for every kind of cache capacity and amount of cache levels for every different system. Also this makes a lot of sense due to modern systems having RAM, which would be used by a cache oblivious algorithm as some type of additional cache level. [3] It is basically the same model as the DAM. But we assume an ideal cache with unknown M and B:

1. Tall cache assumption: $M = \Omega(B^2)$
2. The cache is fully associative: Each line can be loaded into any location in the cache.
3. The replacement policy is optimal.

An illustration can be seen in figure 2 below.

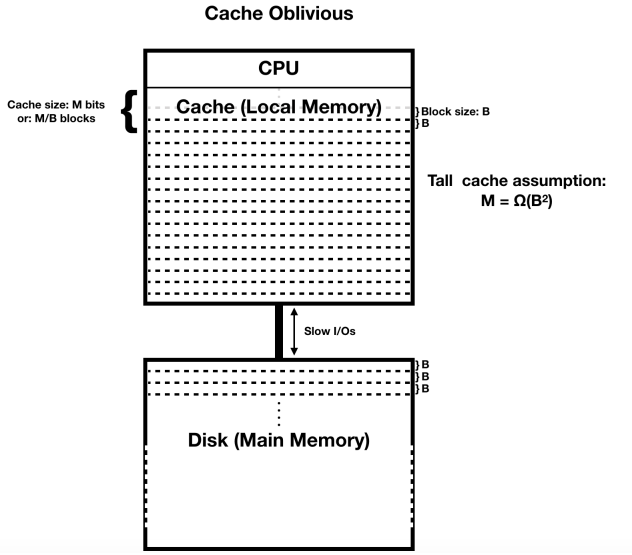

Figure 2: Cache oblivious model with tall cache assumption.

4.3 The dictionary problem [1]

> "A dictionary maintains a collection of key-value pairs $S \subseteq U \times V$, under operations insert(x, v, S), delete(x, S), and query(x, S), which returns the value corresponding to x when $x \in S$."

4.4 The bounds [1]

The paper states the following bounds as a tradeoff between insertions and queries for the dictionary problem in external memory:

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

5 The state of the art

Currently the dictionary problem in external memory is solved with many different data structures. The paper mentions the B^ε-tree, write-optimized skip list and cache-optimized look-ahead array (an optimal version of the LT-LSM). They also bring up the Log structured merge tree, from which they build their data structures. I think additionally we need to mention the Buffer-tree and the B-tree, as they are being used for analysis and comparison in the paper and its references.

5.1 The IP hash table [1, 4]

The original authors Iacono and Patrascu of Theorem 1 also come up with a complicated dictionary solution, which is in the paper referred to as the IP hash table. [4] It requires $O(\frac{1}{B}(\lambda + \log_{M/B} N + \log\log N))$ I/Os for insertion and $O(\log_\lambda N)$ I/Os for queries w.h.p. Therefore we can easily see that if we try to maximize the λ to achieve a lowest possible bound, we can do so until $\lambda = \Omega(\log_{M/B} N + \log\log N)$, because otherwise we would worsen the runtime of the insertions. So the IP hash table is already optimal for such λ. The IP hash table is a construct, which consists of a buffer tree with "gadgets" at the leaves and in addition a "log" array (an array containing all elements in the order of insertion). The idea of the structure is similar to the buffer trees they use: Defer updates (deletes and inserts) to a later time and only write full blocks, so it is possible to cope with the external memory nature of the problem and no I/Os are wasted. The gadgets are basically an optimization technique, but they are complicated, as they consist of multiple fields with different functionality and are recursive.

5.2 The log-structured merge tree (LSM) [1]

The LSM is the foundation of the BOA. The idea is again similar to the buffer tree, where we want to defer updates for a later time to make use of the full B for a memory transfer. The LSMs are not optimal, but will later lead to an optimal structure. There are two slightly different versions of the LSM. The LT-LSM and ST-LSM. As we will see, the ST-LSM is almost like multiple LT-LSMs next to each other. Both consist either of sets of B-trees or sorted arrays, which are called "runs". In the paper, they choose the runs for their structure. Stored within those runs are the key-value pairs and possibly an upsert-message. This is a special message, indicating the deletion of a key-value pair. Thus, it needs to retain temporal order when merging. From this follows that query times are dependent by the number of updates (insertions and deletions) of key K. This is called the duplication count D_K.

5.2.1 The level-tiered LSM (LT-LSM) [1]

An LT-LSM consists of multiple levels of runs. Each level has at most one run. Each level is λ times bigger than the level below (indices-wise, not graphically), which may also be called the growth factor. When a level reaches it's full capacity it is merged to a higher (indices-wise) level, which may cause merges up until the highest level (level $\log_\lambda N - 1$). The amortized insertion time in terms of I/O is low, because the runs are sorted and thus sequential merging is fast. Queries are slower, because depending on the application we can't simply return the first matching element (and therefore most recent) found. Instead we may have to return all the elements on each of the $O(\log_\lambda N)$ levels.

An illustration can be seen in figure 3 below.

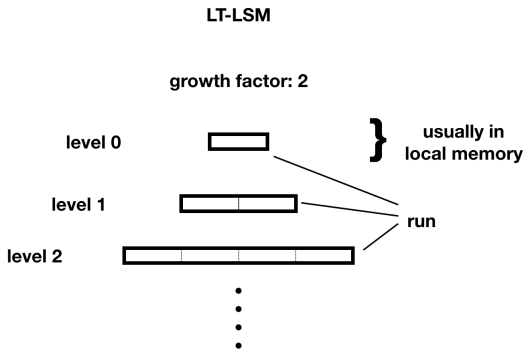

Figure 3: ST-LSM with $\lambda = 2$

5.2.2 The size-tiered LSM (ST-LSM) [1]

Size-tiered LSMs are just like LT-LSMs, except they allow at most $\lambda - 1$ runs per level. The size of the runs scale just as in the LT-LSM: Each level's runs are λ times bigger than the level below.

An illustration can be seen in figure 4 below.

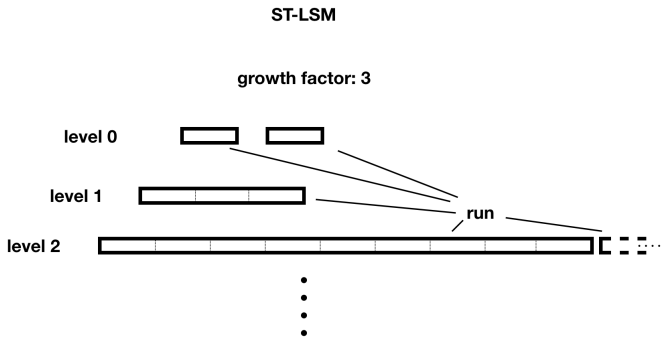

Figure 4: ST-LSM with $\lambda = 3$

6 The Bundle of Arrays Hash Table (BOA) [1]

The BOA is built in the paper using multiple steps to explain the process. First they start with an ST-LSM, where fingerprints are used within the runs. This reduces query time by a factor of $\log N$, but it is still suboptimal according to theorem 1, as it is only optimal in expectation and not w.h.p.:

Note: Lemma 7 refers here to Lemma 1.

Corollary 1 *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 $O(D_K \lambda \log_\lambda N)$ I/Os by writing the levels as in Lemma 7. The insertion/deletion cost is unchanged: $O(\frac{1}{B}(\log_\lambda N + \log_{\frac{M}{B}} N))$ amortized I/Os.*

Lemma 1 *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 $O(N/B)$ I/Os so that membership in A can be determined in $O(1)$ I/Os with high probability.*

The Lemma is proven with a Chernoff-type bound, Bonferroni's inequality and the $\Theta(\log N)$-wise independence of the fingerprints: If we throw N balls (keys) into $\Theta(N/\log N)$ bins (fingerprints) uniformly and $\Theta(\log N)$-wise independently, every bin has $\Theta(\log N)$ balls w.h.p. We can divide the keys of A into N/B buckets, where $B = \Omega(\log N)$ and therefore every bucket contains $\Theta(B)$ keys w.h.p. So this means if we query for a key, we can compute it's fingerprint and therefore find out which bucket it belongs to in $O(1)$. Then we can fetch that bucket in $O(1)$ and scan through the bucket for the right key-value pair for free, because we are in the disk access model. This trick is being used for the BOA and BOT and it is why we use fingerprints instead of keys. Note, that it is not clear how we arrive at the fingerprints. In the full version it simply says, that it is assumed to already be hashed that way. If we had to do that additionally, we would probably have different runtime.

Now that we know why we use fingerprints and that they work, we won't have to worry about them anymore and can use them as a black box as a replacement for keys to achieve better performance at the lower levels.

What is left to do now is optimization at a higher level. As a next step, they introduce routing filters. The goal is to improve query performance so we get the bound of Corollary 1 w.h.p.

Each level gets a routing filter F_l, of which an illustration is in figure 5 below. This is an array where a certain prefix of every fingerprint corresponds to an entry of F_l. The content of that entry is the index of the most recent run, where said fingerprint has been inserted. Now before we query the runs on level l, we query the routing filter, by checking the $F_l[P_l(K)]$ and finding there the index j (the index of the most recent run), for which the prefix $P_l(K)$ of the fingerprint K matches the prefix stored in that run $R_{l,j}$. As we might want every entry and not only the most recent, we chain the runs to a linked list with a "previous field" of an additional character, which contains the previous run with the same prefix. Now we might get false positives from the filter, because fingerprints could have matching prefixes and we also need more space. The paper shows, that if the size of the prefix h_l grows by one character per level, the BOA is optimal according to Theorem 1. This means the size of the routing filter is multiplied by λ at each level (starting from size B), because the array consists of λ^{h_l} characters and $h_l = \log_l B + l$.

So in the end we arrive at the following intermediate, suboptimal result:

Lemma 2 *A BOA supports N insertions and deletions with amortized per entry cost of $O((\lambda +$*

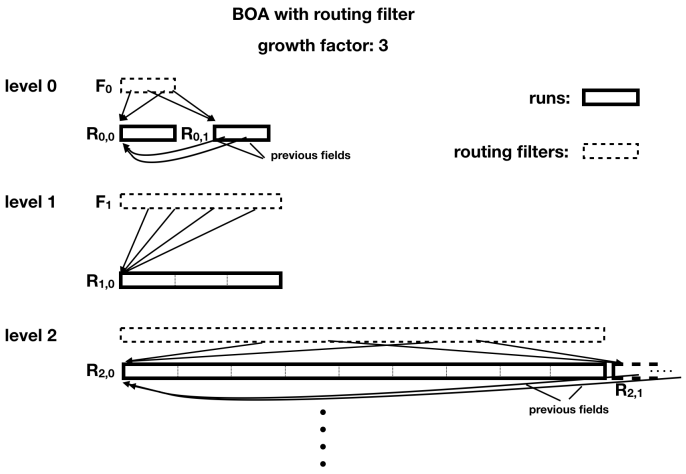

Figure 5: BOA with $\lambda = 3$. Note that the routing filter is not refined yet.

$\log_{\frac{M}{B}} N + \log_\lambda N)/B)$ *I/Os, for any* $\lambda > 1$. *A query for a key* K *costs* $O(D_K \log_\lambda N)$ *I/Os in expectation, where* D_K *is the duplication count of* K.

Now to get from a bound in expectation to a bound w.h.p., we decrease the probability of false positives with an additional check character $C_i(K)$, which is the i-th character from the (right) end of the string representation of the fingerprint K. The paper refers to the pair of array pointer (pointer to the j-th run) and the check character as a sketch. If we think for a moment about inserting fingerprints, we realize that each $F_l[P_l(K)]$ could now have multiple different values for each check character. So when we query the filter at the current level, we get a whole list of possible sketches. To stay efficient, we require each pointer to have $O(1)$ characters per fingerprint as before, but now we may have $\Omega(\log N)$ bits. To do so, we use by $log_\lambda \log_\lambda N$ characters shorter prefixes, called pivot prefixes, and sorted, delta encoded lists of sketches at each entry of the filter array. Delta encoding encodes only the difference of one entry to the previous one, so we store the pivot prefix in the first entry and in the following only a very small number of characters. This results in the following bounds:

Lemma 3 *A refined routing filter can be updated using* $O(\frac{\lambda \log \lambda}{B \log N})$ *I/Os per new entry, and performs lookups in* $O(D_K^*)$ *I/Os w.h.p., where* D_K^* *is the number of times* K *appears in the level.*

And finally it follows what we wanted to reach in the first place:

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

And with a similar observation as with the IP hash table, a BOA is optimal for large enough λ, namely $\lambda = \Omega(\log_{\frac{M}{B}} N + \frac{\log N}{\log \log N})$

7 The Bundle of Trees Hash Table (BOT) [1]

For reference, here is an example of a BOT in figure 6 below.

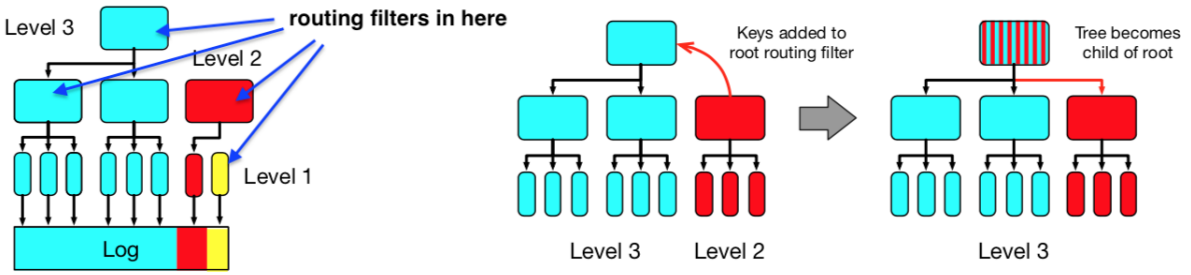

Figure 6: BOT with three levels and $\lambda = 3$. The right part shows an overflow of the second level.

Now that we have reached the bounds of Theorem 1 with the refined BOA, we want to be even faster for insertions. If we think about what we are doing in the BOA most of the time, we realize it's a λ-way merge sort. So to improve, we could try to increase on the arity of that merge sort, which is done in the BOT.

The Bundle of Trees hash table is a bit different from the BOA. It still has levels, but there are no runs. Instead, the fingerprint-value pairs are stored in a single, giant log in the order of arrival. The levels are organised differently. As the name already indicates, instead of a single routing filter we have a whole tree of routing filters at each level (the routing tree). The roots of the trees have a degree of at most $\lambda - 1$ and the degrees of the inner nodes have degree λ. As we insert in arriving order into the log, we append at the right side of the log. Each leaf points to a block of B fingerprints in the log. There are $\lceil \log_\lambda N/B \rceil$ levels. The highest level has a height-$\lceil \log_\lambda N/B \rceil$ tree that indexes the first part of the log (the oldest). For each lower level, a part more to the right (more recent) of the log is indexed.

The routing within the tree happens as follows: Each node of a routing tree consists of a routing filter like in the BOA. This filter is queried as it was in the BOA which results in a list of sketches. The difference now is that they don't point to a run, but to a child node within the routing tree, according to their location in the log. This can only work if we merge the levels in the right way:

The levels are merged as soon as the degree of a node exceeds its limit as in figure 6 on the right side. The routing filter of the root of the concerning tree is merged into the routing filter of the root of the tree of the next higher level. This makes that tree a child of the tree that we merged into, which increases its degree again and could result in further merges.

Now we can't just simply merge a lower level into higher, because in level l we have sketches with prefixes of length h_l and therefore in level $l + 1$, we are lacking a character for the prefix. So to avoid this, each level has character queue in addition, where we retrieve the missing character for the merge. We are not completely happy yet with this solution, because it can still be optimized. A tradeoff arises between merging frequently, which is beneficial to delta-encoding, and defering the merging to increase the arity of the merge. To deal with this, a merging schedule is introduced, which combines the two in the right way and also deals with runs that are shorter than B.

This version of the BOT is not yet satisfactory. The routing tree introduced new ways of having false positives. If we think of how queries are performed (independently) at each level, we see that every node in the tree could generate a false positive. So we need to do some work to get the following strong guarantees:

Lemma 4 *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* $\frac{1}{\lambda}$.

To arrive at such guarantees, it suffices to extend the sketches by a "next character". They consist of the next character which follows the prefix, meaning that we can "peek" onto the prefix of the next level. The idea is that we can now only have the same set of false positives in a child as in the parent, because we can also check the next character. We look at every sketch for which we have a matching prefix and check character. Then we look at the next character of these sketches. If it matches it's parent, it is allowed to stay in the list and we route further down the tree. Otherwise it gets deleted.

The BOT is now complete and has the following properties:

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

Corollary 2 *Let* $\mathcal{B}$ *be a BOT with growth factor* λ *containing* N *entries. If* $\lambda = \Omega(\log_{\frac{M}{B}} N + \log\log M))$, *then* $\mathcal{B}$ *is an optimal dictionary.*

8 The cache oblivious Bundle of Trees Hash Table (COBOT) [1]

A BOT can be made cache oblivious by the following changes: Merging the character queues with funnels, which is cache oblivious. The log has to be buffered into sections of (reasonable) constant size instead of $O(B)$ and items have to be added immediately to the routing filter to eliminate I/Os with optimal caching. At insertion, the fingerprint-value pair is immediately appended to the log and inserted into level 1, which leads each leaf of the routing tree to point to a single entry. The series of character queues have to be placed carefully back-to-back for all j. They are merged using a partial funnelsort (a cache oblivious external merge sort).

As a result we have almost the same bounds as the BOT:

Theorem 4 *If* $M = \Omega(B^2)$, *then a COBOT with* N *entries and growth factor* λ *has amortized insertion/deletion cost* $\Theta(\frac{1}{B}(\lambda + \log\log M + \log_{M/B} N/B))$. *A query for key* K *has cost* $\Theta(D_K \log_\lambda N)$, *w.h.p., where* D_K *is the duplication count of* K.

So it is optimal for $\lambda = \Omega(\log\log M + \log_{M/B} N/B)$.

9 Discussion

9.1 Reaching the lower bound

The authors of the paper have reached the bounds of Theorem 1, which really is tight and not easy to achieve. If one day in-memory comparisons are available these bounds would have to be challenged again.

The authors more or less kept their promise, that their data structure is simpler. Although it is not simple either. Especially the character queue and merging schedule, which I only mentioned briefly. Still it is simpler than the IP hash table they compare it with.

It is a bit sad, that they don't say anything about how they arrive at the fingerprints, as they are crucial for the data structure to work in promised time.

9.2 Possible applications

I can see a wide variety of applications for the BOT. For example I could imagine it to be very useful in a networking scenario where we want fast and reliable key-value store. It would be possible to implement the nodes of the routing tree as nodes in the networks and partitions of the log could be stored separately on different nodes, which would be very nice, because we could route and fetch data all in parallel. On the other hand the nodes could just compute the queries themselves, what would be similar to an in-memory comparison.

Interesting could also be its β-asymmetry, where it is possible to trade writes for more reads, because in a network you could steer traffic and load balance according to the situation.

I don't see it as a replacement for B-trees or similar data structures, because for example a B^+-tree can also support range queries and is a lot simpler. Also the trend nowadays in PCs is to move to SSDs, because in practice with SSDs the read/write latencies are not such strong penalties as they used to be, which lower the cost of random I/O. But if we consider that the lifetime of an SSD is bounded by the number of writes, an argument speaking in favor of the BOT would again be the β-asymmetry, where we can try to write as little as possible. Still, this would have to be compared to other data structures, as they may write even less, but be less efficient. Thus a different model and background would have to be analyzed.

If the COBOT can actually hold it's promised runtimes in practice has yet to be tested, as experiments have not been part of the paper.

References

[1] Alex Conway, Martín Farach-Colton, Philip Shilane. *Optimal Hashing in External Memory*, URL: `http://drops.dagstuhl.de/opus/volltexte/2018/9043/` Digital Object Identifier: 10.4230/LIPIcs.ICALP.2018.39

[2] A. Aggarwal and J. S. Vitter. *The input/output complexity of sorting and related problems.* Commun. ACM, 31(9):11161127, 1988.

[3] Harald Prokop. *Cache-Oblivious Algorithms* Masters thesis, MIT. 1999.

[4] 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 Dis- crete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012, pages 570582. SIAM, 2012. URL: `http://portal.acm.org/citation.cfm?id=2095164&CFID=63838676&CFTOKEN=79617016`, doi:10.1137/1.9781611973099.