B-epsilon-tree and cache-oblivious lookahead array: a comparative study of two write-optimised data structures

Yevhen Khavrona
University of Twente
The Netherlands
o.khavrona@student.utwente.nl

ABSTRACT

The ever-growing amounts of data stored in the world require efficient and fast data structures to store and process it. Due to the large size of such massive data sets, the data structures that operate on them grow so large that they can no longer fit in main memory. Thus, the number of I/O operations between fast main memory and slow disk becomes the performance bottleneck of these data structures. To properly assess their performance, these data structures are analysed in the external memory model that puts emphasis on the number of blocks transferred between main memory and disk. Multiple data structures and their variations were developed in the external memory model to optimise the number of block transfers, among which the B-tree is the most well-known one. One of the research areas related to designing data structures in the external memory model has been focused on making data structures that keep the same search performance as the B-tree but asymptotically improve the speed of writes. Despite extensive theoretical results in the area, little experimental data about performance of such write-optimised data structures is available. In this research study, we analysed two write-optimised data structures - the B^ϵ-tree and cache-oblivious lookahead array (COLA) - and performed experiments to determine which data structure performs better under which conditions. As our results show, the COLA has much better write speeds than the B^ϵ-tree when inserted elements are not sorted, but achieves worse results when the data is sorted. Point queries are faster in the B^ϵ-tree, which makes it a better choice for workloads that require more querying than updating data. Lastly, the support of an efficient read-and-update operation and more streamlined experience of implementing the B^ϵ-tree compared to the COLA make it an even more favourable data structure to consider for a use in data storage systems.

Keywords

External memory (I/O) model, write-optimised data structures (WODS), B^ϵ-tree, cache-oblivious lookahead array (COLA)

1. INTRODUCTION

Ever since the first computers were invented, there has been the need to efficiently store and process stored data. With the emergence of more advanced computer systems and deeper integration of technologies into people's lives, more data than ever needs to be stored and maintained. Market intelligence company IDC reported that around 64.2 zettabytes (1 zettabyte equals 10^{21} bytes) of data was created or replicated in 2020 and the worldwide storage capacity reached 6.7 zettabytes [8].

Such tremendous amounts of generated data necessitate research on space-efficient and fast data structures for massive data sets. Since these data structures are meant to work with data sets of sizes that far exceed the amount of memory available on a computer, the bottleneck in execution time of operations on these data structures is not the speed of a CPU performing instructions but the time spent on transferring blocks between main memory and disk. Thus, data structures that work with massive data sets are analysed in the so-called external memory model (or the I/O model) introduced by Aggarwal and Vitter in 1988 instead of the classic RAM model [1]. In the external memory model, performance of a data structure is evaluated by counting the number of memory-disk block transfers required to perform an operation.

The B-tree is the most famous external memory data structure introduced by Bayer and McCreight in 1970 [2], long before the external memory model was invented. The B-tree generalises the idea behind the binary search tree and adapts it to the external memory model [7].

Despite having optimal query speeds, the B-tree is not an optimal data structure in terms of writing performance. To address this weakness of the B-tree, research efforts have been focused on creating data structures that can keep the same optimal read speed as the B-tree but improve the write speed to be asymptotically faster. Moreover, faster writes not only affect writing performance, but also allow for faster creation and maintenance of search indices, which in turn increases the speed of searches in databases as well [4]. Additionally, increased write speeds help utilise modern flash memory devices better due to inherently slower write speeds of these drives compared to their read speeds [12].

In this research study, we performed a comparative evaluation of two write-optimised data structures - the B^ϵ-tree proposed by Brodal and Fagerberg [5] and cache-oblivious lookahead array (COLA) proposed by Bender at al.[3]. Both data structures employ the same key idea of batching data updates together to reduce the number of block transfers required. However, they differ significantly in the way they realise this idea in practice: the COLA maintains a sequence of arrays geometrically increasing in size while the B^ϵ-tree is an extension of the B-tree that additionally allocates space for update messages in internal nodes.

35^{th} Twente Student Conference on IT July 2^{nd}, 2021, Enschede, The Netherlands.

While the theoretical framework for B^ϵ-trees and COLAs has been established, there is not much comparative information about how such array-based and B-tree-based approaches compare on practice. Thus, in this research study we analysed the theory behind the two data structures, implemented them and investigated their experimental performance to determine the implications for the real-world applications of the two data structures.

According to our results, both data structures are considerably harder to implement than the regular B-tree. However, this additional implementation complexity appears to be worth the effort since the B^ϵ-tree and COLA showed strong gains in terms of increased write speeds in our experiments. The COLA outperformed both B^ϵ-tree and B-tree in the random insertions test. The trees showed a considerable increase in write speeds when inserted data was sorted. As expected, the search speed of COLA was worse than of both the B-tree and B^ϵ-tree.

As a result of our study, we can conclude that the B^ϵ-tree is a more versatile and flexible data structure that has a more streamlined implementation and less space demands.

The rest of the paper is organised as follows: in the second section, we describe the general idea behind B-trees, B^ϵ-trees and COLAs, in the third section we compare the way operations are performed on the latter two structures, in the fourth section we outline the implementation details of the data structures and in the last section, we show the results of an empirical evaluation of the B^ϵ-tree and COLA.

2. PRELIMINARIES

2.1 Search tree in the I/O model

Most commonly, data structures and algorithms are analysed in the RAM model by counting the number of CPU operations that are required to perform a certain action. However, when a data structure grows too large, such approach ceases to show the true performance of the data structure. As such data structure cannot fully fit into memory, its parts have to constantly be swapped in from a slow disk to memory or swapped out back to the disk. In such circumstances, the cost of I/O operations required completely outweighs the cost of computations that take place in the main memory.

Thus, the external memory model filled the void in theoretical performance evaluation of such "large" data structures by switching focus to the number of block transfers that take place between memory and disk. The complexity of algorithms in the I/O model is expressed in terms of the (disk and memory's) block size B, memory size M and the number of elements stored in a data structure N. For the purposes of theoretical analysis in the model, the time taken by computations that happen in main memory is considered to be negligible and thus is not included in analysis.

Regular binary search trees that are commonly used for lookups do not perform well in such a model since their operations are not tuned to optimise the number of I/Os. To address this issue, the idea of a binary search tree was extended to the external memory model and resulted in the creation of the B-tree. The B-tree was invented by Bayer and McCreight in 1970 with the purpose of efficient management of large voluminous indices for random access files [2]. Its design ensures that the number of I/Os that are required to perform operations on the tree stays small compared to a traditional binary search tree. Due to the considerable performance gains of B-trees, they became a de facto standard in modern databases and file systems that have to deal with massive amounts of data [9].

Instead of having only two children as in a binary search tree, the B-tree's fanout is set to be a multiple of the block size B. Similarly, the size of each node of a B-tree is set to be equal to $O(B)$. In a standard B-tree, key-value pairs are stored both in internal nodes and leaves in sorted order. Keys in internal nodes serve as pivots that ensure the sorted order of the tree and guide traversals of it.

Over the years, many variants and implementations of B-trees have been developed, the most widely used of which is the B+ tree [10]. In the B+ tree, only leaves store key-value pairs while internal nodes contain pivots that are used for navigation in the tree. An example of a B+ tree is shown in Figure 1.

2.2 B^ϵ-tree

The B^ϵ-tree is an extension of the B-tree. It's a write-optimised B-tree that was proposed to demonstrate the trade-off curve between external memory data structures that support fast queries and those that support fast updates [4, 5].

Similarly to the B+ tree, the B^ϵ-tree stores key-value pairs in leaf nodes and pivots for navigation towards leaves in internal nodes. Both internal and leaf nodes have size $O(B)$. However, besides storing pivots, internal nodes also store update messages in buffers that are the key to B^ϵ-tree's enhanced write performance. $O(B^\epsilon)$ space is reserved for pivots and children pointers, and $O(B - B^\epsilon)$ is left for the message buffer. A schematic representation of B^ϵ-tree's internal node is depicted in Figure 2.

Instead of directly propagating insertions, deletions and updates down the tree towards target leaves as in the B-tree, in the B^ϵ-tree, these operations are encoded as update messages that are put into internal nodes' buffers starting from the root node. Messages are stored in buffers sorted by the key and creation timestamp (to maintain the order of messages related to the same key).

Only when there are enough update messages in a buffer to move them down efficiently (i.e. when the buffer is full), they are flushed in a batch one level down the tree. Such a strategy ensures that at least $O(\frac{B-B^\epsilon}{B^\epsilon}) = O(B^{1-\epsilon})$ messages are moved together in a single batch [4]. Moving these messages in batches results in fewer I/O transfers than if each individual update was flushed directly to its target leaf. Eventually, each update message will reach its target leaf and will be applied to it.

The position of a specific variant of the B^ϵ-tree on the above mentioned trade-off curve depends on the choice of parameter ϵ that determines how much space in each internal node is reserved for pivots and how much for messages. Depending on the choice of ϵ, the B^ϵ-tree can approximate any structure along the trade-off curve, including a regular B-tree (if $\epsilon = 1$) [4, 5].

When $\epsilon = 0.5$, the B^ϵ-tree achieves asymptotically better write speeds than the B-tree while maintaining comparable read speeds [4]. Read operations keep the same asymptotic complexity and still require $O(log_B N)$ I/Os. However, the amortised write speed of such a B^ϵ-tree increases to $O(\frac{log_B N}{\sqrt{B}})$ compared to $O(log_B N)$ I/Os of the B-tree due to the fact that messages are flushed in batches of size at least $O(\sqrt{B})$. This combination of factors makes such a configuration the most interesting one by far. Therefore, in our experiments, we set ϵ to approximately 0.5.

It's important to stress that the complexity analysis of

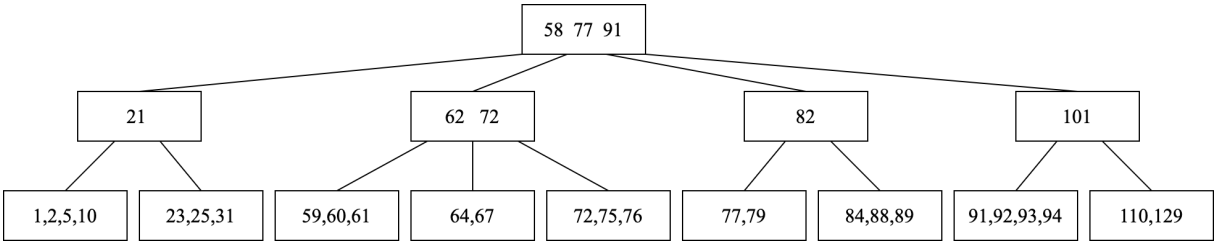

Figure 1. A B+ tree with fanout $F = 4$ and block size $B = 4$. Elements in internal nodes are pivots and elements in leaves are keys.

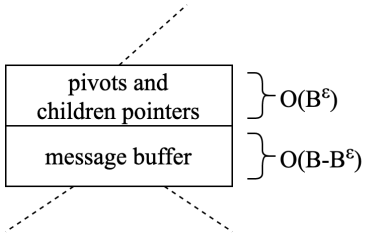

Figure 2. An internal node of a B^ϵ-tree with $O(B^\epsilon)$ space reserved for pivot-children pairs and $O(B - B^\epsilon)$ space for the message buffer.

updates in the B^ϵ-tree is amortised since some updates might trigger recursive flushing of messages down the tree, thus increasing the I/O cost of that single update.

2.3 Cache-oblivious lookahead array (COLA)

Cache-oblivious lookahead array (COLA) is a write-optimised data structure that is a variation of the log-structured merge-tree (LSM-tree) proposed by Bender at al. [3]. LSM-trees are write-optimised data structures, first described by P.O'Neil et al. in 1996 [11], that cover a range of multi-level data structures, each level of which is larger than the previous one by some multiplicative factor G. LSM-trees have faster write speeds than B-trees since their pattern of growth allows for batching of updates in a similar fashion to B^ϵ-trees.

While LSM-trees typically use tree-like data structures to represent levels, COLA uses sorted arrays that are stored contiguously on disk [3]. In its basic version, the COLA scales by a growth factor of 2, i.e. each subsequent array is twice larger than the previous one, and thus such a 2-COLA has $\lceil log_2 N \rceil$ levels in total. In 2-COLA, each level is either full or empty. The kth level of 2-COLA is full if the kth least significant bit of a binary representation of the number of elements in COLA N is set to 1. When there is not enough space in existing arrays, the COLA creates a new array that is twice larger than the previously largest array and moves all elements in a batch into the new array. These batched movements of keys make sure that the cost of updates is asymptotically better than in B-trees.

The COLA keeps the same asymptotic read speed as B-trees by applying fractional cascading introduced by Chazelle et al. to speed up key searches [6]. To find a key in a COLA, each level has to be searched, but running binary search on each individual level results in an asymptotically worse query complexity than in the B-tree. To address this issue, in a COLA, each 8th element of the $(k + 1)$st array is copied to the kth array with a lookahead pointer to its position in its original array. Each fourth spot in an array is reserved for a duplicate lookeahead pointer that points to the closest real lookahead pointers to its left and right [3]. Such technique allows to run only a single binary search and follow it up by a sequence of constant-sized scans in subsequent levels. The 2-COLA that illustrates the idea of fractional cascading is shown in Figure 3.

Originally, the COLA was designed as a cache-oblivious data structure, i.e. it does not need to know the block size B for tuning its operations. However, the COLA can be turned into a cache-aware lookahead array with similar complexity bounds to those of the B^ϵ-tree by setting the growth factor G to $O(B^\epsilon)$ and including each $O(B^\epsilon)$th element of array $(k + 1)$th array as a lookeahead pointer in array k [3]. These changes allow COLA to have faster queries that match the ones of the B^ϵ-tree while sacrificing some writing speed.

Similarly to the B^ϵ-tree, the performance analysis of insertions and deletions in COLA is amortised as some updates might cause expensive rebuilding of arrays of the data structure. With the help of additional buffers per level, the COLA can be deamortised and offer better complexity guaranties per an individual update [3].

2.4 Operations on data structures

2.4.1 Insertions and deletions

In the B^ϵ-tree, insertions differ significantly from insertions in the regular B+ tree. Instead of propagating the key down the tree, an update message with the inserted key is put into the buffer of the root node. If the buffer of the root node fills up, a batch of messages is flushed down to either one or more of the root's children [4]. If the child's buffer is (almost) full, its messages are also flushed to its children in batches. Such policy ensures that after a certain number of flushes each message is delivered to a correct target leaf node. Similarly to the B-tree, if a leaf receives too many keys, it splits. If an internal node receives too many children (pivots), it splits and distributes the pivots and messages from its buffer to the newly created internal node.

Since each insertion goes through $O(\log_{\sqrt{B}} N)$ levels of the tree (when $\epsilon = 0.5$) until it eventually reaches the target leaf and messages are flushed in batches of at least $O(\sqrt{B})$ messages, the amortised insertion cost is $O(\frac{\log_B N}{\sqrt{B}})$ block transfers [4].

Multiple policies for flushing messages and keeping buffers can be created. For instance, the child with the largest amount of pending messages can be selected to flush messages to [4]. The buffers might be kept without a specific number of message slots reserved for each child or they might allocate exactly $O(B^{1-\epsilon})$ space for each child's messages and allow flushing in batches of exactly $O(B^{1-\epsilon})$ messages.

When a key needs to be deleted from the tree, a delete

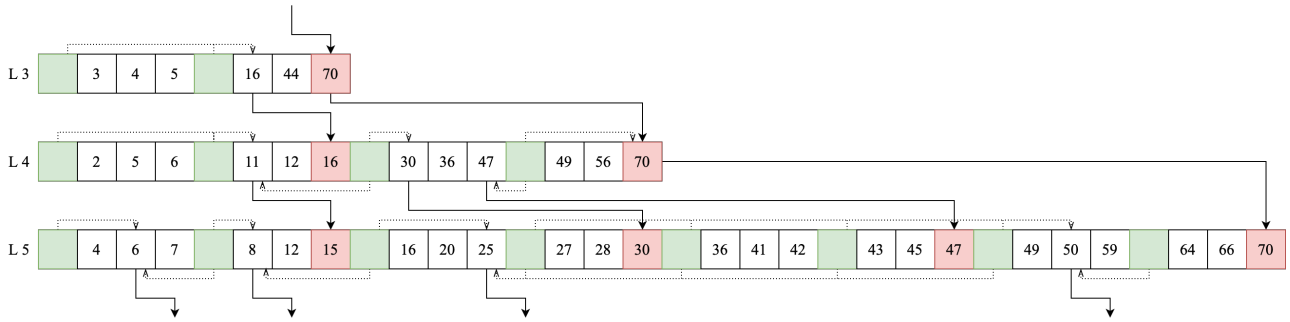

Figure 3. Here only levels 3, 4 and 5 of a 2-COLA are shown. The rest of the COLA's levels are omitted for brevity. Red cells contain keys that are selected to be inserted into preceding arrays as lookahead pointers and green cells mark spots reserved for duplicate lookahead pointers. Solid arrows represent lookahead pointers from array k to the subsequent array $k+1$ and dashed arrows show duplicate lookahead pointers that point to closest lookahead pointers.

message with this key's value is inserted into the root [4]. Then, the procedure continues in the same way as for updates until the message eventually reaches its target leaf node. Since insertions and deletions are algorithmically similar, their I/O complexity is the same.

Insertions in the COLA (with $G = 2$) start with insertion of a key into a special buffer that can hold precisely one element. Then, if there is already a level of size 1, the buffer is merged with that level into the following level of size 2. These merges into larger levels proceed until no new merges are required and the element is put into its target array. In the worst case, the inserted key has to go through $O(\log N)$ merges before being inserted into the target array. In order to merge two arrays of size k, $O(k/B)$ block transfers are needed, where B is the block size. Therefore, $O(1/B)$ block transfers are spent per each item, which leads to the total amortised cost of insertion $O(\frac{\log N}{B})$ I/Os [3].

After the merging procedure is finished, the lookeahead pointers that were present in the merged arrays before are no longer valid. Thus, they need to be redistributed from scratch starting from the target level and continuing down level by level, until the first level that is set to contain lookeahead pointers is reached. Asymptotically, the cost of insertion still stays at $O(\frac{\log N}{B})$ block transfers.

In the cache-aware version of COLA, each level is smaller than its subsequent level by a factor of $O(B^\epsilon)$, which means that before the level k becomes full, the level $(k+1)$ has to be merged into it $O(B^\epsilon)$ times [3]. Since there are $O(\log_{B^\epsilon}(N))$ levels in total, the cost of insertion into such a COLA is $O(\frac{\log_{B^\epsilon}(N)}{B^{1\epsilon}})$ I/Os.

Deletions in the COLA can be implemented by employing techniques used by other variants of LSM-trees and B^ϵ-trees, e.g. by performing only a logical deletion of a key with a tombstone mark without actually deleting it from the structure.

Overall, in theory the COLA can offer higher insertion speeds than the B^ϵ-tree because of division by the factor $O(B)$ instead of $O(\sqrt{B})$ as in the B^ϵ-tree, unless the size of the B^ϵ-tree node is chosen to be large.

2.4.2 Point queries

Since insertions, deletions and updates are scattered around the nodes of the B^ϵ-tree, the point query procedure is more complicated than the one of the ordinary B-tree [4]. However, the guarantee that all updates to a leaf node are located on the path to that node allows searches in the B^ϵ-tree to have the same optimal I/O complexity as in the B-tree.

Searching starts from the root by checking the root's buffers for update messages [4]. If an insert or delete message is found, the search can stop. If there is an update messages, it has to be carried further and applied to any other update messages found along the path to the target leaf. If the search hasn't stopped at the root, it continues performing the same actions recursively on the correct child node (that is chosen according to the pivots stored at the root) until the search reaches the target leaf. Finally, if the leaf is reached, its keys are scanned to find the key. In the worst case, each query has to go down $O(\log_{\sqrt{B}} N))$ levels of the B^ϵ-tree with $\epsilon = 0.5$ to reach a leaf node, thus leading to the same query complexity of $O(\log_B N))$ I/Os as in the B-tree.

For the COLA, in the worst case each array has to be searched to find a key [3]. With the help of fractional cascading and lookahead pointers, only the initial binary search is necessary which is then followed by a scan of a constant number of keys in each subsequent array. Thus, instead of $O(\log^2 N)$ I/Os in the case of performing $O(\log N)$ binary searches, a point query incurs only $O(\log N)$ I/Os in the worst case.

Therefore, the query cost of the 2-COLA is slightly worse than the one of the B^ϵ-tree due to the difference in the base of the logarithm, so it's expected that queries in a 2-COLA are slower than queries in the B^ϵ-tree.

However, the speed of queries in COLA can be increased by making it cache-aware according to the procedure described before. In such a case, the base of the logarithm in query cost increases due to a larger growth factor and smaller number of arrays, which leads to faster query speeds of $O(\log_B N)$ I/Os.

2.4.3 Upserts

One major advantage of the B^ϵ-tree is its support of a special type of operation - an upsert [4]. An upsert represents a typical workload in a database by combining two common operations into one - querying data and performing updates based on the result of the query. Since search speeds of both COLA and B^ϵ-tree are far worse than their write speeds, searching data before performing an update would cancel all the benefit from asymptotic optimisation of updates. Therefore, it's vital in such a case to perform an upsert without the need to query data first. While the structure of COLA does not present an obvious way to support a fast upsert, the message-based nature of the B^ϵ-tree allows for easy extension of its operation range to

include upserts. Upserts in B^ϵ-trees are simply encoded as one more type of update messages that includes the (pointers to) actions that have to be performed on the key if it's found in the tree. Since upserts in B^ϵ-tree do not require prior searches, their cost remains bounded by the cost of a write.

Asymptotic complexities of operations on the two data structures are summarised in Table 1.

2.5 Space requirements

The basic slow version of COLA without lookahead pointers requires $O(N)$ of contiguous space on disk. When lookahead pointers are added to a 2-COLA, its space requirements grow twofold to $O(2N)$ [3]. In a G-COLA, space complexity of the data structure depends on the sampling density of lookahead pointers. In case of a deamortised COLA, the space it takes grows even further by a constant factor to cover the additional buffers at each level. Such space requirements are more demanding than the ones of the B^ϵ-tree, that keeps space close to $O(N)$ without large constant factors, and can make the B^ϵ-tree a more appealing choice if space is an important consideration.

3. IMPLEMENTATION

We implemented both B^ϵ-tree and COLA in C++. Block transfers between disk and memory are automatically managed through memory mapping by the operating system. We map a large file on a disk into memory and work directly with the mapped memory.

Both data structures are implemented in their simplified form, i.e without the support for variable-length keys, heuristic-based optimisations or other features that would be important in practice. However, our implementation covers all the vital details of the data structures and is sufficient to test the theoretical concepts in question.

For the COLA, we roughly followed the implementation described by its authors [3]. We implemented a non-amortised version of COLA that can be tuned with the growth factor G and pointer density PD. Parameter PD represents the number of lookahead pointers that each level contains, i.e. if $PD = 0.5$, each array level besides k regular keys holds $0.5k$ lookahead pointers sampled from the subsequent level. The last level does not contain lookahead pointers.

As in the original paper, in our implementation keys and values each have the size of 8 bytes. Also, instead of using duplicate lookahead pointers, in each key-value pair we store a copy of the closest lookahead pointer to the right of it. The closest lookahead pointers to the left are determined when the subsequent level is scanned since the distance between two lookahead pointers in their array of origin is known based on G and P. Each lookahead pointer consists of an 8-byte key and 8-byte index of its position in its origin array. Real keys use 8 bytes of padding while lookahead pointers use 16 bytes. All elements are stored right-aligned in their levels.

Since the results of array merges can be too large to fit into memory similarly to the arrays themselves, they have to be written to disk as well. To save on additional disk space, we follow the strategy of merging outlined in the original paper: the result of the first array merge in insertion is placed into the rightmost position in the target array, then for the second merge, the result is placed into the beginning of the mapped region which has just been freed up due to a previous merge. For the subsequent merges, we continue with alternating between the two destinations. As one additional element spot is required when merging is performed, we keep a buffer for that newly inserted element.

Invariants of the 2-COLA about level fullness and size do not hold in the G-Cola. Some levels in the G-COLA might contain only lookahead pointers and no real keys. Each level in the G-Cola might be full or have a size that is a multiple of previous level's size. We use these facts to determine the number of elements present in each level of COLA and the way to distribute lookahead pointers.

To create a cache-aware version of COLA, we set the pointer density PD to 1 as such a setting corresponds to sampling $O(B^\epsilon)$ elements from array $(k+1)$ to array k when $G = O(B^\epsilon)$.

Our implementation of the B^ϵ-tree closely follows the theoretical description of the data structure. In all our experiments with the B^ϵ-tree, we set ϵ to 0.5. In each internal node, there are slots for pivots and children pointers with the rest of node's space reserved for the message buffer. Each pivot-pointer pair takes 16 bytes of space. Our implementation supports only insert messages as they are representative enough of the other message types as well. Each messages takes 32 bytes and consists of a key, value, timestamp and type fields. Message buffers are implemented as arrays. Additionally, each internal node stores metadata about the number of pivot-pointer pairs it contains, the number of messages in its message buffer and its offset from the beginning of the mapped region.

Leaves have all $O(B)$ space reserved for key-value pairs. To match the implementation of COLA, keys and values take 8 bytes of space each. Leaf nodes store as metadata the number of keys they contain and their offset from the beginning of the mapped region.

As in the theoretical model of the B^ϵ-tree, in our implementation a key is inserted as an insert message into the root first. If the root's buffer fills, the child with the most pending insertions is selected for flushing to. If the selected child's buffer contains too many messages to accommodate the flushed batch from the root, the flushing process continues recursively from the child. The child node flushes batches to its children until it can accommodate the batch from its parent.

Internal nodes in our implementation only flush message batches that exceed a certain threshold. The threshold is set to the ratio between the number of messages in a node's buffer and the node's maximum fanout (i.e. the maximum number of children a node can have). If there is no batch of size larger than the threshold to flush, the node splits.

When a leaf node is reached, the insertion encoded in messages are applied to the child. If a leaf receives to many insertions, it splits into two leaves and distributes its keys evenly. If an internal node gets too many pivots, it also splits into two nodes and distributes pivots, children pointers and messages between the two nodes.

Besides the two write-optimised data structures, we have also implemented a regular B+ tree to serve as a baseline for our experiments. As with the B^ϵ-tree, we followed the textbook description of the B+ tree closely. Each node of our B+ tree has size 4096 bytes, and keys and values in the tree take 8 bytes of space each.

Overall, we found both B^ϵ-tree and B+ tree to be easier to implement in code as the theory behind both structures aligns better with practice than the theory behind the COLA. In terms of lines of code, both B^ϵ-tree and COLA reached around 1000 lines of C++ code while B+ tree

DS	Insertion	Point query	Upsert (search followed by update)
B-tree	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N)$
B^ϵ-tree	$O(\frac{\log_B N}{\epsilon B^{1-\epsilon}})$	$O(\frac{\log_B N}{\epsilon})$	$O(\frac{\log_B N}{\epsilon})$
B^ϵ-tree, $\epsilon = 0.5$	$O(\frac{\log_B N}{\sqrt{B}})$	$O(\log_B N)$	$O(\log_B N)$
2-COLA	$O(\frac{\log N}{B})$	$O(\log N)$	$O(\log N)$
G-COLA	$O(\frac{G \log_G N}{B})$	$O(\log_G N)$	$O(\log_G N)$
Cache-aware LA	$O(\frac{\log_B N}{B^{1-\epsilon}})$	$O(\log_{B^\epsilon} N)$	$O(\log_{B^\epsilon} N)$

Table 1. I/O complexity of operations on B^ϵ-trees and COLAs

required less than half of that amount. Implementation-wise, the B^ϵ-tree and COLA were considerably harder to implement than the regular B-tree.

4. EMPIRICAL EVALUATION

4.1 Setup

We ran our experiments in a sandboxed virtual machine to restrict the influence of external factors on performance of the data structures. We used VirtualBox v6.1.22 virtual machine running Ubuntu 20.04.2.0 LTS with a single core of a 2.9GHz quad-core Intel Core i7 CPU (with 8MB shared L3 cache) and 3GB of RAM available. A single core ensured that all operations on data structures were performed sequentially. For the backing disk, we used Transcend StoreJet 25M3 external HDD. The system's page (block) size was set to 4096 bytes. The default Linux swap file was turned off before running the experiments.

Before the testing started, we allocated a large file on the disk that would serve as a backing storage for the data structures. Before each test, we flushed the memory buffers to remove cached pages, remounted the disk and remapped the disk file into memory.

With the help of the C++ Boost library, for each of our experiments we measured the wall clock time and CPU time (subdivided into the user time and kernel time). As we were interested in the I/O performance of the data structures, we followed the approach described in [3] to estimate the time spent on block transfers: we measured the disk time as the difference between the wall clock time and CPU time reported. The time not spent in user- or kernel-space is mostly devoted to running I/O operations, so such a way of measuring time enabled us to observe the amortised I/O performance of the data structures.

4.2 Experiments and results

First, we inserted around 2^{25} random key-value pairs into the B+ tree, B^ϵ-tree, 4-COLA with pointer density $PD = 0.1$, 32-COLA with $PD = 1$ and 64-COLA with $PD = 1$.

Figure 4 demonstrates that the increased write speeds of the COLA in theory are clearly reflected on practice as well. The 4-COLA achieves at least 10-times better write speeds than the B+ tree and several times faster writes than the B^ϵ-tree when the structures fill the memory. The COLA is significantly faster for random insertions since

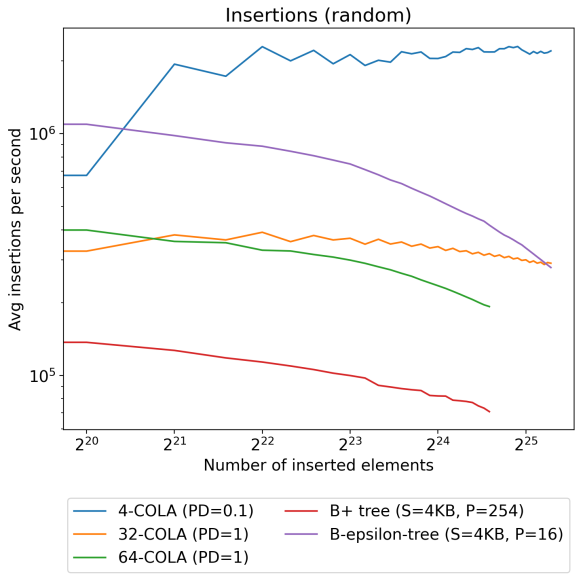

Figure 4. Random insertion of key-value pairs. All graphs are plotted on the logarithmic scale. PD stands for pointer density, S for node size and P for the number of pivots.

its logarithmic cost of insertions is divided by the factor of $O(B)$, even though the base of the logarithm is smaller than that for the B^ϵ-tree and B+ tree. Even when the B^ϵ-tree and B+ tree start slowing down, the 4-COLA doesn't yet show signs of decreasing write speeds and has a much flatter downward-trending curve. The non-smooth wavy appearance of the COLA's graph is caused by level merges. In the beginning, multiple new levels are created, so the total I/O cost of these operations appears to be higher on the graph and thus the initial speed of insertions of the 4-COLA is slightly smaller than of the B^ϵ-tree.

The B^ϵ-tree shows a better write performance than the regular B+ tree due to the usage of update messages and batched updates. For each single insertion, the I/O cost is shared with a number of other insertions. As expected, the B+ tree has the worst random insertion performance as its write speed is not optimal.

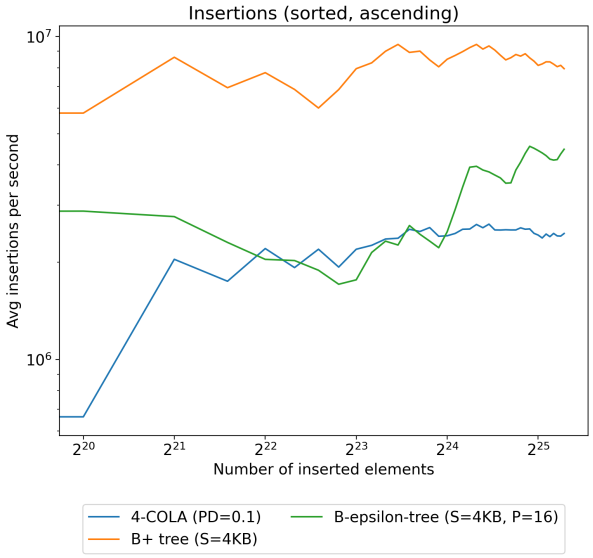

Figure 5. Sorted insertion (ascending) of key-value pairs

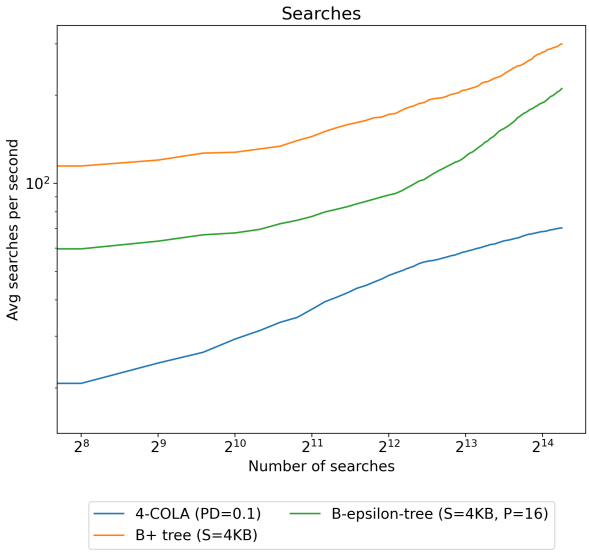

Figure 6. Point queries (searches)

Both 32-COLA and 64-COLA perform worse than the 4-COLA in terms of insertion speed. The more we increase the growth factor G in a COLA to be closer to B (i.e. by making the COLA cache-aware), the slower insertions become as merging larger arrays requires a larger amortised number of I/O transfers. The insertion pattern of the 32-COLA is more similar to the one of 4-COLA while the curve of the 64-COLA (which growth factor is closer to B^ϵ) already resembles the one of B^ϵ-tree.

Overall, it's observable that for random insertions the Cola has a clear (asymptotic) advantage over both B^ϵ-trees and B+ trees.

After testing random insertions, we inserted about 2^{26} sorted key-value pairs into the three data structures in ascending order. Figure 5 shows the result of the test.

It can be observed that for sorted insertions the situation is considerably different than for the random ones. The COLA can no longer maintain the same advantage in performance over the B^ϵ-tree and B+ tree. However, the reason of a change in the graphs lies within the bump in performance of the trees, not in the slower write speeds of the COLA. Sorted insertions are the best case for the B^ϵ-tree and B+ tree since they let the trees to solely use either the leftmost part of the tree (in case of descending insertions) or the rightmost part (in case of ascending insertions). As only these parts of the tree need to stay in memory, there are less block transfers involved compared to random insertions.

The results of insertion experiments show that neither the B^ϵ-tree nor COLA is consistently faster across the range of different workloads. As we showed, both data structures have their strengths and weaknesses depending on the specifics of data they work with. However, if we assume that the typical workload of a database is concerned more with randomised insertions, the trees will not enjoy the same advantage as they showed in the last two experiments and the COLA will be a much faster option.

Next, we experimented with point query characteristics of the data structures. The results are presented in Figure 6. First we inserted about 2^{27} keys into a B+ tree, 4-COLA and B^ϵ-tree. For all data structures, the results show that the more nodes (parts of arrays) are loaded into memory and cached, the faster searching becomes. For searches, the B-tree is faster than the other data structures, proving why it still stays the most popular data structure for database design. 4-COLA is the slowest out of the three data structures with several times slower query speeds than in the B-tree.

As for space requirements, the COLA with fractional cascading took more space than the trees as its arrays grow geometrically by a factor of G each time. Since we kept elements in arrays right-aligned according to the described implementation, each time space had to be allocated for the whole array, even if only a small portion of it was filled at that time.

5. CONCLUSION

Write-optimised data structures have a huge potential to increase performance of databases and file systems not only through increased speeds of writing data but also through the faster creation of indices used for searching. Both B^ϵ-tree and COLA seem to be good choices for these purposes, although each data structure has its own advantages and disadvantages. While COLAs work better for random insertions, they tend to have slower reading speeds than B-trees. B^ϵ-trees, on the other hand, have slower writing speed than COLAs but maintain better reading speed. As we showed in our experiments, the COLA can be tuned with the block size B to become a cache-aware lookahead array that approximates the performance of the B^ϵ-tree.

In general, for environments with a substantial demand for writing data, such as the messaging applications, the COLA might serve as a better choice. For more balanced environments or for environments with more demand for reading data, we would prefer the B^ϵ-tree.

The B^ϵ-tree involved less implementation complexity and aligned better with theoretical description on practice than the COLA. Moreover, the B^ϵ-tree's support of upserts is

one of its strongest sides that allows it to keep fast write speeds even if there is a need to perform a search operation before the write one. The COLA's performance in such a case degrades significantly and effectively cancels all the benefits of write optimisation. Lastly, the memory footprint of the B^ϵ-tree is smaller than the one of (deamortised) COLA that might be an important consideration in space-limited environments.

6. FUTURE WORK

There are many areas to which the research about write-optimised data structures can be directed. For the cache-oblivious lookahead array, its space requirements and a way to make it more space-friendly can be investigated. For the B^ϵ-tree, a variety of message-flushing policies can be looked into to determine the most optimal one. Although some optimisations to increase the data structures' performance for specific workloads were presented in [3] and [4], there are might be more heuristics and optimisations to uncover and experiments with.

7. REFERENCES

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

[2] R. Bayer and E. McCreight. July 1970. *Organization and maintenance of large ordered indices. Mathematical and Information Sciences Report*, 20.

[3] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92, 2007.

[4] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. An introduction to b -trees and write-optimization. *Operating Systems and Sysadmin*, page 22, 2015.

[5] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. Citeseer, 2003.

[6] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[7] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[8] IDC. Data creation and replication will grow at a faster rate than installed storage capacity, according to the idc global datasphere and storagesphere forecasts. https://www.idc.com/getdoc.jsp?containerId=prUS47560321. Accessed: 2021-04-30.

[9] V. Jain, J. Lennon, and H. Gupta. Lsm-trees and b-trees: The best of both worlds. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1829–1831, New York, NY, USA, 2019. Association for Computing Machinery.

[10] H. C. V. Ngu and J.-H. Huh. B+-tree construction on massive data with hadoop. *Cluster computing*, 22(1):1011–1021, 2019.

[11] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[12] T. I. Papon and M. Athanassoulis. The need for a new i/o model. In *CIDR*, 2021.