Universität Hamburg
Department Informatik

Wormhole: A Fast Ordered Index for In-memory Data Management

Betreuerin: Kira Duwe

Efficient Programming

Markus Heidrich
Matr.Nr. 6719523
markus.heidrich@informatik.uni-hamburg.de

29.02.2020

Abstract

Large data sets are more and more common in today's age of big data and cloud computing. To work fast with that data, and leverage the computing power one has, in-memory data management systems such as key-value stores are often used. To support range queries ordered indexes must be used. Most of the ordered indexes used today though, like B+ tree, have a look-up cost of $O(log\, N)$, where N is the number of keys in an index. With billions of keys, this look-up time is one of the limiting performance factors.
In the Paper I discuss here, a new ordered index structure called $Wormhole$ is introduced, that combines the strengths of a hash table, prefix tree and B+ tree, and thus achieves a lookup time of $O(log\, L)$ in the worst case for looking for a key with length L.
"Experiment results show that Wormhole outperforms skip list, B+ tree, ART, and Masstree by up to 8.4, 4.9, 4.3, and 6.6 in terms of key lookup throughput, respectively."[1]

Contents

1 Introduction

By removing expensive I/O operations and hosting data in memory, accessing the data becomes a major cost factor, "reportedly consuming 14-94% of query execution time in today's in-memory databases"[1].
By optimising hash tables, one can push the index performance to close to the hardware's limit[1]. The resulting $O(1)$ lookup performance is however not available for range queries, that ordered index structures like B+ tree with a lookup performance of $O(log\, N)$ support[1]. "When the B+ tree grows to billions of keys, ... on average 30 or more key-comparisons are required for a lookup"[1].
Another approach for an ordered index is prefix tree, or also known as trie. By ordering keys through their content instead of their relative position in the key set, trie achieves a search cost of $O(L)$, where L is the length of the keys[1]. Trie is thus able to outperform B+ tree for example, at least when the keys are not too long, as its performance does not depend on the amount of data in the key-value store. Still, "with its issues of inflated index size and fragmented memory usage, trie has not been an index structure of choice in general-purpose in-memory data management systems"[1].
By combining the B+ tree with a trie to replace the non-leaf section, Wormhole is able to remove the factor N in its lookup time and replace it with $O(L)$. Further combining the resulting structure with a hash table reduces the cost to $O(log\, L)$[1]. All common index operations, like lookup, insert, and range query, are efficiently supported[1]. The goal in the following is to give an intuitive understanding to readers, of how Wormhole works.

2 The Wormhole Data Structure

The idea of the construction of the Wormhole structure starts of with a B+ tree.

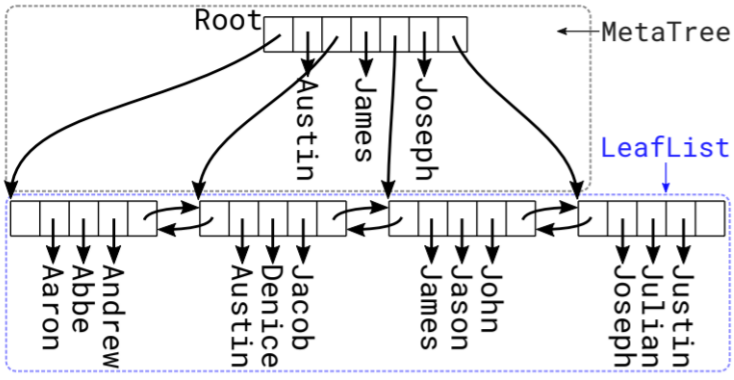

Figure 1: An example B+ tree.[1]

Here we call the internal tree MetaTree, and the connected leafs LeafList[1]. Now, a B+ trees search performance of $(O(log\, N))$ is dependent on the number of

keys in it, as the internal MetaTree growth logarithmically the more keys are in the LeafList, and more comparisons are necessary to walk through MetaTree[1]. The first optimisation we take a look at is replacing MetaTree with a structure where the search cost is not tied to the number of keys N in the LeafList.

2.1 Replacing the MetaTree with a Trie

A tries search cost ($O(L)$) is bound to the length of the search key L. By choosing good keys, one can thus easily outperform a B+ tree. That is why we now replace MetaTree with a trie.

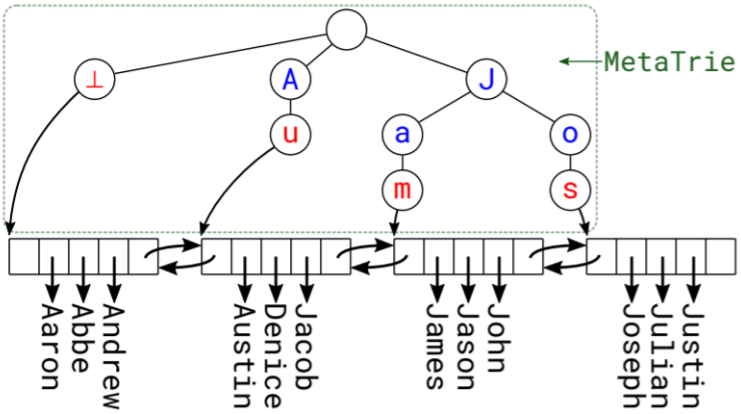

Figure 2: Replacing B+ trees's MetaTree with MetaTrie.[1]

Figure 2 shows the MetaTrie that replaces MetaTree. By choosing for each leaf node a key as its anchor, and inserting the anchor in the MetaTrie, a borderline between a node and the node to its left is created (assuming the sorted LeafList is laid out in ascending order, as shown in Figure 2.)[1].
There are two conditions for an anchor key ($anchor-key$) of a node ($Node_b$). The first one is the ordering condition:
Any key on the left of the $anchor-key$ is smaller then the $anchor-key$. Any key "bellow" or on the right of the $anchor-key$ is equal to or greater than the $anchor-key$. The result is that all leaf nodes left of $Node_b$ contain keys that are smaller than the keys of $Node_b$ when compared with one another, and all keys and nodes on the right of the $anchor-key$ and the $Node_b$ are greater[1].
The second condition is the prefix condition: "An anchor key cannot be a prefix of another anchor key"[1]. This means that no branch may point to more than one node, as is seen in Figure 2.
To make the trie as small as possible, only the longest common prefix plus an additional two tags get inserted into the MetaTrie as an anchor for another leaf node, thus fulfilling the prefix condition. The $\perp$ in Figure 2 denotes the smallest tag and is used to implement the prefix condition[1].
The idea how the MetaTrie generally works is what we look at here, so we ignore

the specific use of the smallest token, as it is not important in understanding the idea of the MetaTrie in general. The smallest token gets added to a prefix to avoid conflicts with the prefix condition, but is ignored for the ordering condition.
The lookup cost can now be considered $O(L)$ in the worst case, where L is the length of the key. Walking the MetaTrie is what incurs this cost. Searching on the leaf nodes can be considered constant or $O(1)$ because of their fixed maximum size.

2.2 Performing Search on MetaTrie

Searching on the MetaTrie works more or less like a regular trie structure. Tokens in the key get matched to those in the trie one at a time until a leaf is reached. Unlike a regular trie, however, not every key is in the trie. An example would be the key "Denice" in Figure 2, already the matching of token "D" fails[1].
To work around this issue, Xinbo Wu et al. introduce the concept of a target node for search key K. "A target node for K is such a leaf node whose anchor key K1 and immediately next anchor key K2 satisfy K1 $\leq$ K $<$ K2, if the anchor key K2 exists. Otherwise, the last leaf node on the LeafList is the search keys target node."[1] This means that for key K the anchor of the nearest node to its left in the MetaTrie is smaller or equal to K, and the anchor key of the nearest leaf node to its right, if it exists, is greater than K. While searching for a key, the MetaTrie is walked on until one reaches the longest common prefix. Now, we know that the anchor keys to the right of the next tag of the search key are larger than the search key. Meaning, if we walk on the MetaTrie to the nearest right anchor key, the search key must be in the node left to the node we reached. If we instead follow the anchor key that is immediate to the left of the tag after the longest common prefix of the anchor key, we know that all keys in the leaf node reached here are less or equal to our search key. The search key thus must be in the node reached, if it exists.
The following examples, "A", "Denice" and "Julian", are explained in more detail.

When searching for "Julian" the longest common prefix is "J", as can be seen in Figure 3. The next tag is "u", not in the MetaTrie, and has no anchor key to its right, but one to its left. Following that anchor key to the leaf node, we know that "Julian" must be in the node we reached, as the search key is greater than the anchor key "Jos". In a linear search, we find "Julian" in the node.
When searching for "Denice", we already fail to find the tag "D", but find two tags on its sides. "A" is less than "D", thus following this anchor key leads to the node the key must be in. Or we follow the tag to its right, that starts with "J" and is thus greater than the search key. From the tag "J" we have two options to reach an anchor key. To reach the correct node, we always follow the left-most tags when walking to the right from the search key. The anchor key we find here leads us to the beginning of a node where every key is greater than the key we search for. Accordingly, we search to the node left to the one we reach and find "Denice" there. The two walk options can be seen in Figure 3 by following the green arrows.

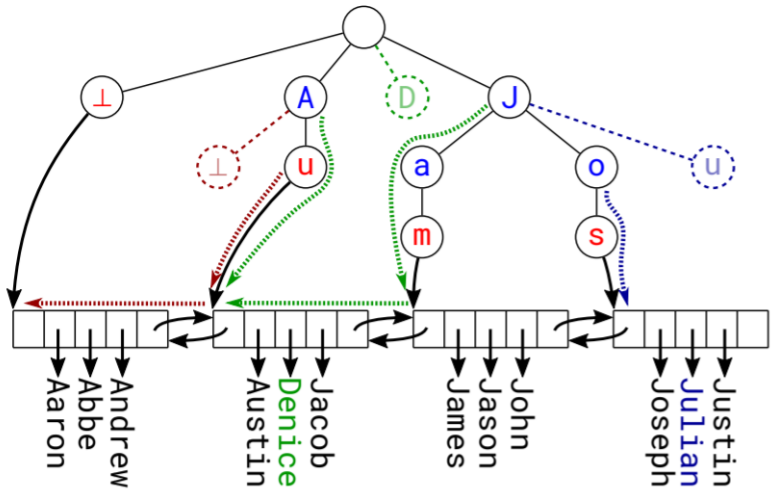

Figure 3: Replacing B+ trees's MetaTree with MetaTrie.[1]

While searching for the key "A" we match the only tag it has. Because of the prefix condition, and "A" having only one token, we now expect the smallest token as an option to reach from "A", as seen in Figure 3. As this is not the case, we walk the only option there logically is: The tag to the right of the smallest token we expected leads us to the anchor key "Au". This anchor key is greater than the search key "A", and accordingly, we search the node to the left of the one we reached, and fail to locate the key "A".

2.3 Accelerating Search with a Hash Table

To further increase the speedup archived untill now from $O(L)$ to $O(\log L)$, Xinbo Wu et al. replace the MetaTrie with a hash table, MetaTrieHT.

All prefixes of the anchor key get inserted into the hash table MetaTrieHT. For anchor key "Jos" this means inserting "J", "Jo" and "Jos", as can be seen in Figure 4[1]. Additionally we have several fields, that come in useful while searching for the correct leaf node.

An example to try to search for is "Jason". Through remembering the length of the longest anchor key, denoted L_{anc}, we are able to find the longest common prefix "Ja" in $O(log(min(L_{anc}, L_{key})$[1]. This is done, amongst other things through the help of the field with the bitmap. This field contains a bit for every sibling node, and is searchable in $O(1)$ time[1].

As "Ja" is not an anchor key, described by the field "Internal or Leaf" in Figure 4, we need to continue searching for the correct leaf node. Looking at the bitmap field we try to find the nearest right or left sibling of the next tag "s" of the search key. In this case, only the left sibling "m" is found. Now we search "Jam" in the MetaTrieHT, and see that it is denoted as a leaf node, and accordingly an anchor key. The fields "Left-most Leaf" and "Right-most leaf" are identical and point both to leaf node 3. Because we followed the left sibling of "s" we know that every

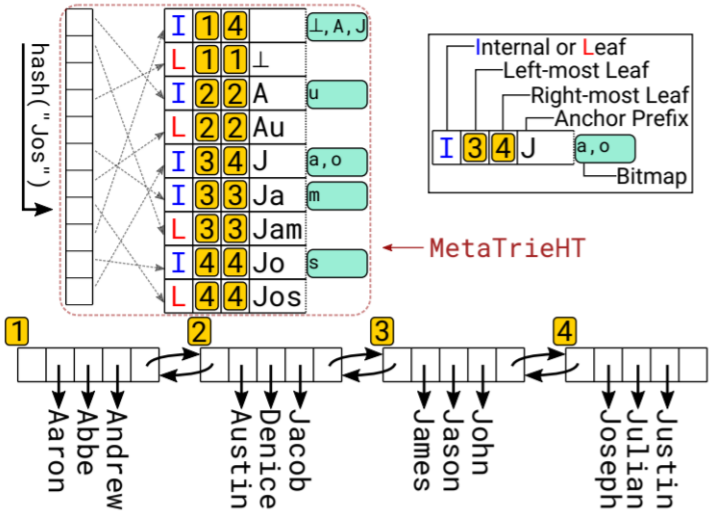

Figure 4: The structure of Wormhole. The bitmap lists its child nodes.[1]

key in leaf node 3 is smaller or equal to our search key "Jason", and search in this node for the key. If there was a right sibling and we followed that one, in the same principle as with the MetaTrie, we would have reached a leaf node that only contains keys that are greater than "Jason". Accordingly we would search in the node to the left of the one we reached.
Lookup, insertion, deletion and range query now have a time cost of $O(log\, L)$. Wormhole is asymptotically faster than B+ tree and trie, with search costs of $O(log\, N)$ and $O(L)$ respectively, where N is the number of keys and L the search key's length[1].
Insertion and deletion function similar to insertion and deletion in a B+ tree. Leaf nodes get merged or split as needed, and instead of propagating these changes through the internal nodes of a B+ tree, anchor keys get inserted or deleted in the MetaTrieHT[1].

3 Parallelism

Scalability needs the ability to perform multiple operations at once, which is why Wormhole provides concurrency support. This is done by minimizing the use of locks, and their impact[1].
One can differentiate 3 different groups of access exclusiveness that Wormhole requires[1]:
The first one requires no access exclusiveness. As long as no data is inserted or deleted, this is the access the operation has.
The second group requires access exclusiveness of one or multiple leaf nodes, because of inserting or deleting in them.
The third group demands the exclusiveness of the MetaTrieHT and the effected leaf nodes. This occurs when an insertion or deletion incurs a merge or split of the

leaf nodes and therefore modifies also the MetaTrieHT.
To realise this, Xingbo Wu et al. use two different kinds of locks. One is a read writer lock for the leaf nodes. This lock corresponds to the second group of access exclusiveness. Only one key is inserted or deleted, and accordingly, only one leaf node is locked and unavailable for lookup[1].
The third group requires the other lock type. Here, the two leaf nodes that merge or split are locked. Additionally, a single mutex lock on the entire MetaTrieHT is granted to remove or add an anchor and its prefixes[1].
Obviously, every operation requires access to the MetaTrieHT, and locking MetaTrieHT will cost performance when no other operation can access it. That is why operations that perform read-only access should still be able to operate to enhance performance, as they themself do not change the Wormhole.
To this end Xingbo Wu et al. use the following technique: An operation where a split or merge occurs, acquires a lock and applies its changes to a second hash table (T2), an identical copy of MetaTrieHT (T1). T1 is still accessible for its readers. When the changes are applied to T2 it becomes visible for its readers through updating the pointer to the current MetaTrieHT, and T1 becomes invisible for new readers. After waiting for all operations to finish that still use T1, the same changes are applied to it(T1).
The additional space used by the second MetaTrieHT is according to Xingbo Wu et al. negligible.
The question now is, how does a lookup guarantee that the hash table that was used is consistent with the leaf node? Merged or split leaf nodes are released as soon as that part of the operation is finished, they do not wait for the update of the MetaTrieHT[1]. The solution presented in the paper is version numbers to check for consistency. The version number of the MetaTrieHT is incremented whenever a split or merge occurs, and the affected nodes are set to the version number of the MetaTrieHT. An operation remembers the version of the MetaTrieHt it used, and compares it with the number of the leaf node. If the expected version of the leaf node is less than the one it actually has, the lookup aborts and starts over.
"The penalty of the start-overs is limited."[1]

4 Evaluation

In the paper, Xingbo Wu et al. "experimentally evaluate Wormhole by comparing it with several commonly used index structures, including B+ tree, skip list, Adaptive Radix Tree (ART), and Masstree"[1]. While doing so, they try to ensure that through, for example, choosing data sets, how keys are formed, how threads are used, that the comparisons are abt and performance differences are because of differences in the algorithms and not something else.
Specifics will be omitted in the following, like which data set is used, or what implementation of an algorithm.

As is shown in Figure 5, Wormhole with one thread performs about 52% better

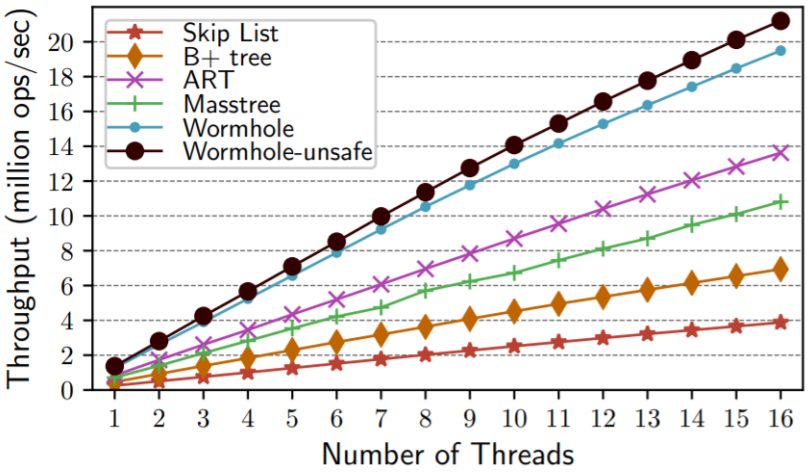

Figure 5: Lookup throughput with different number of threads.[1]

than the next best algorithm, ART. While using 16 threads, Wormholes throughput is about 15.4 times as great as with one thread and continues to outperform all other algorithms, with the nearest one being again ART, with a difference of about 4.5 MOPS.[1]
The thread-unsafe version of Wormhole performs about 7.8% better.[1]

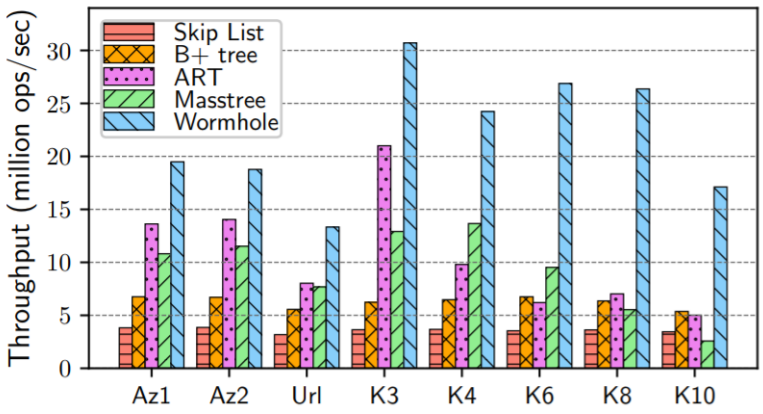

Figure 6: Lookup throughput on local CPU with different data sets and key length.[1]

Figure 6 shows clearly that Wormhole outperforms the other algorithms. That Wormhole has less variation in its performance with keys of different length then other trie based algorithms is explained by its use of the anchor length (L_{anc}), "which is generally (much) smaller than the average key length"[1].
"Wormhole improves the lookup throughput by 1.3x to 4.2x when compared with the best results among the other indexes for each keyset."[1]

The memory usage of Wormhole is in most cases comparable to B+ tree and skip list, as can be seen in Figure 7. As Wormhole "uses a small trie to organize its

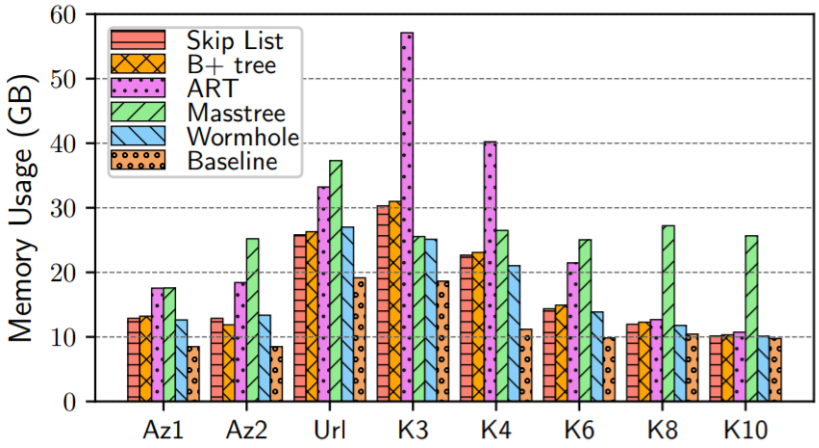

Figure 7: Memory usage of the indexes.[1]

anchors and places the keys in large leaf nodes"[1], it is more space-efficient than trie-based algorithms like Masstree, that place the entire key in the trie.[1]

5 Conclusion

The problem Wormhole addresses, accessing data of key-value stores in-memory fast, is relevant in the age of big data, cloud computing and modern data centres.[1] By combining the strength of B+ tree, prefix tree and hash tables, Wormhole is able to reduce the worst-case lookup cost to $O(log\, L)$, where L is the length of the keys, while still supporting lookup, insertion, deletion, concurrency and especially range queries. The performance of Wormhole is thus independent of the number of keys and affords a substantial speed-up, when the key length is chosen correctly. Due to these improvements, it is not unlikely that Wormhole or something built upon it will be widely applied in the next 5 to 10 years.

References

[1] Song Jiang Xingbo Wu, Fan Ni. Wormhole: A fast ordered index for in-memory data management. March 2019.