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

Wormhole, a new high-performance in-memory data management method, is an ordered index structure that takes a worst case time of $O(\log L)$ to search a key of length $L$. This low cost is obtained by leveraging the advantages of B+-tree, hash table and prefix tree namely, the space efficiency of B+ tree and lookup cost of a trie $O(L)$ and hash table $O(1)$. Initially the B+ tree structure of implementation is considered. Here, the leaf nodes are called as LeafList and all other nodes together are called a MetaTree. The MetaTree has a search time cost of $O(\log N)$, which becomes a problem when the number of keys in the system is too huge. To reduce this time cost, the non-leaf section of the B+ tree is replaced with a trie structure. By doing this, the dependency of $N$ in the B+tree’s search time is removed. This resulting structure is called a MetaTrie. Also, an anchor, which acts as a borderline for two leaf node, is identified for each node in the LeafList and inserted it into MetaTrie.

The MetaTrie has a worst case search cost of $O(L)$ which is comparatively better than the B+tree search cost, if the length of the key is small. The search in a trie is done by matching token in a search key to a token in trie. To reduce the worst case search cost the authors made use of a binary search algorithm, proposed by Waldvogel et al., to perform longest prefix match between a search key and an anchor. Also, the trie structure in the MetaTrie is stored as a hash table, which will index the the leaf nodes in the LeafList. By doing binary search and using hash table, the search cost is reduced to $O(\log L)$. The space cost for Wormhole is also better than trie as multiple keys are stored in a leaf node, rather than individual keys in the trie. This makes indexing faster. When compared to the internal nodes of a B+ tree, the Wormhole’s MetaTrieHT will have lesser number of nodes as it is highly likely to have long common prefixes between adjacent keys in the LeafList and thereby reducing the total number of entries in the hash table. As the LeafList in a MetaTrieHT is always in sorted order, this structure will support range search as well.

(1) Use Figure 5 as an example to explain how a MetaTrieHT is built.

To find a target node, in MetaTrie structure, we have two phases:

1. Conducting the longest prefix match (LPM) between the search key and the anchors in the trie.
2. In case the longest prefix does not match an anchor, we need to do search on a subtree rooted at a sibling of the token next to the matched prefix of the search key.

But each of these phases costs $O(L)$. MetaTrieHT tries to reduce this cost to $O(\log L)$. To reduce the search cost in the first phase, the wormhole implements a binary search on prefix lengths to accelerate the pattern matching. For performing the binary search, all the prefixes of each anchor is inserted into a hash table. For example, for the anchor ‘Jam’ in Figure 1, we insert all of its prefixes and the anchor into the hash table i.e, “ ”,”J”, ”Ja”, ”Jam” is added. Similarly all the anchors in the MetaTrie and its prefixes are added to the MetaTrieHT.

![Figure 1 : MetaTrie with 12 keys](image-url)
Each of the entry in the hash table will have a bit indicating whether the node is an internal node or an anchor node. If the node represents an internal node, the hash item is denoted with ‘I’ and if the node is an anchor or leaf node, it is denoted using ‘L’. For example, ‘A’ in figure 1 will be represented using ‘I’ in the hash table whereas ‘Au’ will be represented using ‘L’.

To support the second phase of the search, each hash item will have extra two fields. The first field is a bitmap, which will contain bits indicating the child nodes. The second field contains two pointers, pointing to the right-most and left-most leaf nodes. That is for the internal node ‘J’, the bitmap field will have ‘a’ and ‘o’ and the second field will have pointer to the leaf nodes 3 and 4 (figure 2).

(2) Assume the anchor “Alexander” is in the MetaTrieHT, and the search key is “Alexandria”. Show how many steps (hash table lookups) are required to complete the first phase.

It is given that anchor “Alexander” is in the MetaTrieHT, that means all of its prefixes will also be present in the MetaTrieHT. So the MetaTrieHT will have “ ”, “A”, “Al”, “Ale”, “Alex”, “Alexa”, “Alexan”, “Alexand”, “Alexande”, “Alexander” as hash items inserted.

First phase is finding the longest prefix match between the search key and the anchor using a binary search algorithm. The total number of steps taken for this phase will be equal to the total number of hash table lookups. For the given search key “Alexandria”:

a. Split the search key in middle and search for the first half in the Trie. i.e, we will check if “Alexa” is present in the trie(hash table). Since it is present we will move to the 2nd half of the key.

b. Now we split the 2nd half of the search key and check if that prefix is present in the trie. i.e, we will check if “Alexand” is present. Since it is present we will move to check the other half.

c. The remaining half is split and compared with the trie. i.e, we compare “Alexandr” with the trie. Since is not present we will return the last found match,”Alexand” as the longest prefix match.
The above figure shows the three hash table lookups done when searching for the longest prefix for the key “Alexandria”.

(3) Use Figure 5 as an example to show how the search key “Jacob” is found in the tree.

The search in MetaTrieHT has two phases:

i. First we will find the longest prefix match between the search key and the anchors using a binary search algorithm.

ii. If the longest prefix match does not match an anchor, the leftmost leaf node of the right sibling or the rightmost node of left sibling is taken to be the target node and then searched for the key accordingly.
Here we are trying to search for the key “Jacob”. For this key, first we will find the longest prefix match as below:

- We will check if “Ja” is present in the trie. Since it is present we will check if “Jac” is present in the trie.
- As “Jac” is not present in the trie, the algorithm will return “Ja” as the longest prefix.
- The hash function will give the index for “Ja” in the MetaTrieHT.

But from the hash item it is seen that “Ja” is an internal node and not an anchor. This means the second phase has to be done. For the second phase we first check the bitmap corresponding to “Ja”. The bitmap has only one entry ‘m’. As ‘c’ is less than ‘m’, we will consider ‘m’ to be the right sibling. As “Jam” is a right sibling for ‘Jac’, and an anchor node, we would take its leftmost leaf node as the target node’s next node, which is a pointer to the node 3 as per the above diagram. And the target node will be the node immediately to the left of the node 3, i.e., node 2, which is reached by walking backward from node 3. Now a straightforward lookup is done on Node 2 to find the key “Jacob”.

References: