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

Wormhole is an ordered index structure that provides high-performance in-memory data management. Wormhole efficiently supports all common index operations, including lookup, insertion, deletion, and range query. It has a lookup cost of $O(\log L)$ memory accesses, where $L$ is the length of search key. It leverages the advantages of three indexing structures namely, space efficiency of B+ tree (by storing multiple items in a tree node), trie’s search time independent of store size (dependent on L), and hash-table’s $O(1)$ search time, to obtain a single efficient index.

The internal node structure storing the subset of the keys in a B+ tree is called a MetaTree and the leaf nodes where the keys are stored is the LeafList. This is replaced by the trie structure in order to remove the $N$ factor in the B+ tree’s $O(\log N)$ search time. This structure is called the MetaTrie which has the lookup cost to $O(L)$. In order to reduce the lookup cost on the trie structure to $O(\log L)$, the hash table replaces the MetaTrie structure and is named MetaTrieHT, to index the leaf nodes on the LeafList. This structure is called the Wormhole.

The MetaTrieHT makes use of the Binary search algorithm to do the longest prefix match between the search key and the anchors in the trie. By doing this search cost is reduced to $O(\log L)$. This index structure is more space efficient by storing multiple keys in a leaf node. It also performs efficient range search as the keys are ordered.

Questions:

1) Show an example B+ tree and an example prefix tree. Do both support range search? For a given number of keys, which one has a lower lookup cost?

Answer:

B+ tree, an ordered index search tree is an extension of B tree. It consists of a root node, internal nodes and leaf nodes. In B+ tree all keys are placed in leaf nodes while internal nodes store a subset of the keys and pointers to facilitate locating search key at leaf nodes. The leaf nodes are linked using a link list; therefore, a B+ tree can support random access as well as sequential access. This also supports fast processing of range-search queries as all the nodes are ordered. The B+ tree is space efficient because a leaf node’s size, or number of keys held in the node, is bounded in a predefined range $[k/2, k]$ ($k$ is a predefined constant integer) the search with a leaf node takes $O(1)$ time. The main disadvantage is the lookup cost $O(\log N)$, where $N$ is the number of keys. For example, a B+ tree hosting billions of keys, might take more than 30 key-comparisons in a lookup, which is more expensive than a hash table.

Prefix tree or a Trie is also an ordered tree data structure that is based on the prefix of a string. Every node of Trie consists of multiple branches and each node represents character in a key. The child node acts as an array of pointers (or references) to next level Trie. The lookup cost of the prefix tree is $O(L)$ where $L$ is the length of the search key,
which is comparatively better than B+ tree. For example, for a Trie where keys are 4-byte integers and each byte is a token, the search cost is upper-bounded by a constant (4) regardless of the number of keys in the index.

Both these tree data structures support Range search as all their keys are ordered. For a given key, the lookup cost is comparatively lower in prefix tree as the cost depends only the length of the key (L) whereas in B+ tree it depends on N, the number of keys in the tree. This is true only when the key size is small. For instance, if the number of keys in B+ tree is 20 and the search key size is 4 bytes, there will be three key comparisons in B+ tree whereas in prefix tree it’s a constant 4 regardless of the number of keys in the index. Suppose now the size of the search key is 30 bytes, B+ tree will be more efficient compared to the Prefix tee. Hence, we cannot conclude which of these tree structure has lower lookup cost as it depends on N or L.

2) Please design a table to compare B+tree, prefix tree, and hash table on their lookup cost, support of range search, and space efficiency.

Answer:

<table>
<thead>
<tr>
<th></th>
<th>Lookup cost</th>
<th>Range search</th>
<th>Space efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>B+ tree</strong></td>
<td>High lookup cost with a large N(Number off keys)</td>
<td>Allows Range search</td>
<td>Space efficient (long arrays)</td>
</tr>
<tr>
<td></td>
<td>O(log N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Prefix tree</strong></td>
<td>High lookup cost even with a moderate L</td>
<td>Allows Range search</td>
<td>Space inefficiency</td>
</tr>
<tr>
<td></td>
<td>(Length of the key) O(L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hash table</strong></td>
<td>O(1)</td>
<td>Unable to perform range operations</td>
<td>Space inefficiency</td>
</tr>
</tbody>
</table>

Table: comparison of B+-tree, prefix tree, and hash table

*Lookup cost:* The B+ tree lookup cost [O(log N)] is high when compared to the prefix tree and the hash table as it depends on the number of keys present, N. Whereas the Trie’s search cost [O(L)] is determined by the number of tokens in the search key (L). If keys are long, even with a small set of keys in the Trie, the search cost can be high. The hash table’s lookup cost is O(1) as each key stored in the table has its own pointer which helps in fast lookup.

*Range Search:* Both B+ tree and the Prefix tree allows range search as they have ordered indexes. Whereas hash table does not support range queries whose search identifiers are not existent in the index.

*Space Efficiency:* The B+ tree is space efficient because a leaf node’s size, or number of keys held in the node, is bounded in a predefined range [k/2, k] (k is a predefined constant integer), the search with a leaf node takes O(1) time. Whereas in Prefix tree each key is stored in a separate node which requires more space as the number of keys increases. The hash table requires an entry (or pointer) for every key in the index, demanding a higher space cost.

3) If we replace B+ tree’s MetaTree with a hash table, what are the issues? Can we have a B+ tree AND additionally a hash table to accelerate lookup at MetaTree?
Answer:

Replacing B+ tree’s MetaTree with the hash table can reduce the search cost to O(1) as the key can be searched in one step if present. However, there are many issues to be considered.

➢ The use of hash table does not support inserting a new key at the correct position in the sorted LeafList which in turn makes it difficult for the range search to be performed.
➢ The space cost is inefficient as the hash table requires an entry (or pointer) for every key in the index.
➢ If the key is not present in the structure then, hash function returns the value that is not present in the hash table. This becomes a problem during the range search when the first and last key of a range is not present. Example: to find a range of (k1 to k5) keys, if k1 and k5 are not present in the hash table, the lookup will not happen.

Yes, it is possible to have a structure of hash table and B+ tree combined. The hash table can have a pointer pointing to the root node of the B+ tree. When there is a lookup, the key will be first searched in the hash table, if not present the search will be performed in the B+ tree. This structure will have the following issues:

➢ Space inefficiency: This structure will take more space as both hash table and B+ tree are stored. This means more memory resource will be used.
➢ Inconsistency: There are high chances that the data in both structures may be inconsistent and might give a false result. But the issue can be solved by locking the system while the operations are taking place.

4) With B+ tree’s MetaTree replaced by a MetaTrie, anchors are inserted into the trie. Use Fig. 3 as an example to explain how an anchor is determined? If the last key in the first leaf node is “Austi”, what’s the anchor between the first and the second leaf nodes?

Answer:
The above figure shows the replacement of B+ tree’s MetaTree with MetaTrie. This is a better structure than MetaTree as the look up cost of a trie is $O(L)$ which is independent of $N$(number of keys). In the LeafList for every node, an anchor is inserted to the MetaTrie which acts as a borderline between this node and the node immediately to its left, assuming the LeafList is horizontal and arranged in an ascending order.

The anchor key of the node should follow two conditions:

a) **The Ordering Condition:** left-key $<$ anchor-key $\leq$ node-key, where left-key represents any key in the node (Node$_b$) immediately left to Node$_b$, and node-key represents any key in Node$_b$. If Node$_b$ is the left-most node in the LeafList, the condition is anchor-key $\leq$ node-key.

b) **The Prefix Condition:** An anchor key cannot be a prefix of another anchor key.

Assume that, Node$_b$ is a new leaf node whose anchor key has not been determined. Following are the steps to determine an anchor for this node.

Let the smallest key in Node$_b$ be $\langle P1P2...PkB1B2...Bm \rangle$ and the largest key in previous node Node$_a$ be $\langle P1P2...PkJA1A2...An \rangle$ and $A1 < B1$.

- If Node$_b$ is not the left-most node on the LeafList ($m > 0$):
  - check whether $\langle P1P2...PkB1 \rangle$ is a prefix of the anchor key of the next node Node$_c$. If not Node$_b$’s anchor is $\langle P1P2...PkB1 \rangle$. Otherwise, Node$_b$’s anchor is $\langle P1P2...PkJ\perp \rangle$, (where $\perp$ represents the smallest token) which cannot be a prefix of Node$_b$’s anchor.
  - check whether Node$_a$’s anchor is a prefix of Node$_b$’s anchor (Node$_a$ is $\langle P1P2...Pj \rangle$, where $j \leq k$). If so, Node$_a$’s anchor will be changed to $\langle P1P2...Pj\perp \rangle$.

- Otherwise (Node$_b$ is the left-most node), its anchor is $\perp$.

Let’s assume in the above figure that Node$_b$’s anchor is yet to be determined. The smallest key of Node$_b$ is “Austin”. As per the ordering condition, the anchor for the Node$_b$ is taken as “Au”, which greater than the largest key of Node$_a$ “Andrew” and lesser than the smallest key of Node$_b$ “Austin”. As Node$_b$ is not the leftmost node on the LeafList, check whether “Au” is a prefix of the anchor key of Node$_c$ and Node$_a$. Since the prefix condition is not violated, “Au” is considered as the anchor for the Node$_b$.

If the last key in the first leaf node is “Austi”, the anchor between the first and the second leaf nodes is determined as follows:

The smallest key of Node$_b$ is “Austin”. As per the ordering condition, the anchor for the Node$_b$ is taken as “Austin”, which greater than the largest key of Node$_c$ which is “Austi” and same as the smallest key of Node$_b$. As Node$_b$ is not the leftmost node on the LeafList, check whether “Austin” is a prefix of the anchor key of Node$_c$ and Node$_a$. Since the prefix condition is not violated, “Austin” is considered as the anchor between the first and the second leaf nodes.

5) Use Figure 4 as an example to explain how search keys “A”, “Denice”, and “Julian” are found in the tree?
Figure 4: Example lookups on a MetaTrie with search keys “A”, “Denice”, and “Julian”.

The basic lookup operation on the MetaTrie with a search key takes place by matching tokens in the key to those in the trie one at a time and walk down the trie level by level accordingly. This leads the lookup to the leaf node in the LeafList where the key is stored. This matching process breaks in one of two situations. The first one is that a token in the key is found to be non-existent at the corresponding level of the trie. The second one is that tokens of the search key run out during the matching before a leaf node is reached. This is because the keys are stored only at the LeafList and are not directly indexed by the trie structure. To address this issue, concept of target node is used for a search key. The target node is identified by taking a path along the immediate left or right sibling. If a search key is in the index, it must be in its target node. The target nodes of “A”, “Denice”, and “Joseph” are the first, second, and fourth leaf nodes in Figure 3, respectively and can be identified as follows:

- **Search key: “A”**
  The key “A” matches the value “A” on the first level of the MetaTrie. But the search process breaks because the tokens of the key run out before reaching the leaf node. To find the target node, we append the smallest token \( \bot \) to the search key. As the token will not be used in the regular key, \( \bot \) becomes the first unmatched key. In this case only the right subtree exists. So, the target node will be first node which is node before the left most node of the right. Now the normal lookup in the target node is performed and found that key “A” is not present.

- **Search key: “Denice”**
  The tokens of “Denice” does not match any value in the MetaTrie. Hence the process breaks, and target node needs to be found. For this, we find the siblings for the token “D” which are subtrees rooted at internal node “A” and “J”. The target node will be the right most leaf node of “A” or the leaf node immediately before the left-most leaf node of the “J”. As the leaf nodes are doubly linked, the target node can be reached by walking backward on the LeafList by one node. Doing the normal lookup in the target node, “Denice” is found as the second key in the second node.

- **Search key: “Julian”**
  For “Julian” the token “J” matches the value “J” in the first level of the MetaTrie. It breaks at the second level of the MetaTrie as the token “u” is not present. Here only the left subtree rooted at internal node “O” is available and only one search path down to the right-most leaf node exists to reach the target node which is the fourth leaf node. By the normal lookup the key “Julian” is found to be the second key of the target node.
References
