Wormhole: A Fast Ordered Index Data Structure

Xingbo Wu

Fan Ni  Song Jiang
The index can be very big.
- The memory becomes increasing large.
- The indexed data (key-value items) can be small.
- Many billions of keys have to be indexed.

The index operations can be very expensive.

“The TPC-H queries spend up to 94% (35% on average) and TPC-DS queries spend up to 77% (45% on average) of their execution time on indexing.”

Kocberber, et al. at IEEE/ACM MICRO’13

The index needs to be sorted.
- Hash table with $O(1)$ operations often isn’t a choice.
- But we will leverage it to accelerate search in a sorted index.
B+-tree Can be too Expensive

- \( O(\log N) \) search cost \((N \text{ is the number of keys})\)
  - \( \sim 30 \) key-comparisons for one billion keys.

- A key comparison may induce multiple cache misses.
- An index search may end up with 50-100 memory accesses.
Prefix Tree Can be Time- and Space-inefficient

- **O(\(L\))** search cost (\(L\) is token count in a search key).
  - 100+ tokens in a key (e.g., URL)

- High space cost due to small node size.
The Dilemma and our Solution

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| B+ Tree        | * Space efficient (large leaf nodes)  
                  * Support of range operations | High lookup cost with a large $N$ |
| Prefix Tree    | Lookup cost not correlated with $N$   | * High lookup cost even with a moderate $L$  
                      * Space inefficiency |
| Hash Table     | $O(1)$ lookup cost                   | No support of range operations |

We will orchestrate **B+-tree, prefix tree, and hash table** in one index structure that:

- has $O(\log L)$ search cost;
- is memory-efficient;
- supports range search.
A Closer Look at B+-tree

The list of leaf nodes:
- All keys in sorted order
- Fast range operations
- Constant lookup cost within each node

The tree of internal nodes:
- $O(\log N)$ lookup cost
- Operations for maintaining tree balance
Replace MetaTree with a Hash Table?

However, this doesn’t work:

- No support of insert;
- No full support of range search.
Replace MetaTree with MetaTrie (a Prefix Tree)

- Keys - at-its-left < anchor ≤ keys-at-its-right - Example anchors: “Au”, “Jam”, “Jos”
- All prefixes of anchors are in MetaTrie.
Search for “James” …….
Search for “Denice” ……
Search for “Denice” …..
Search for “Julian” …….
But the Search Cost is Still $O(L)$

- Two phases in a search:
  - Find the longest prefix match (LPM) on the trie [1st phase].
  - Reach the right-most leaf of the missing node’s left sibling, or the left-most leaf of the missing node’s right sibling [2nd phase].

Binary search on the prefix length ($O(L) \rightarrow O(\log L)$)

Record pointers of left-most and right-most nodes at each trie node ($O(L) \rightarrow O(1)$)
An Example: Binary Search on the Key with a Hash Table

- The anchor is “Alexander”

- All prefixes of the anchor are inserted to the hash table.
  - “A”, “Al”, “Ale”, …, “Alexander”

- Binary search of LPM for key “Alexandria”
  - Len = 5 \( \lceil (0+9)/2 \rceil \): “Alexa” exists.
An Example: Binary Search on the Key with a Hash Table

- The anchor is “Alexander”

- All prefixes of the anchor are inserted to the hash table.
  - “A”, “Al”, “Ale”, ..., “Alexander”

- Binary search of LPM for key “Alexandria”
  - Len = 5 \(\left\lfloor (0+9)/2 \right\rfloor\): “Alexa” exists.
  - Len = 7 \(\left\lfloor (5+9)/2 \right\rfloor\): “Alexand” exists
An Example: Binary Search on the Key with a Hash Table

- The anchor is “Alexander”

- All prefixes of the anchor are inserted to the hash table.
  - “A”, “Al”, “Ale”, …, “Alexander”

- Binary search of LPM for key “Alexandria”
  - Len = 5 \( \lceil (0+9)/2 \rceil \): “Alexa” exists.
  - Len = 7 \( \lceil (5+9)/2 \rceil \): “Alexand” exists
  - Len = 8 \( \lceil (7+9)/2 \rceil \): “Alexandr” does not exist.
  - so LPM is “Alexand” of length 7.
Wormhole:
Replace MetaTree with MetaTrieHT (a Prefix on a Hash Table)
Search for “Jacob” ……

- Search “Ja”
- But “Jac” is not in the table.
- Go to Jam’s left-most leaf node, which is Node 3
- Go to the node’s left sibling, which is Node 2
- Search for “Jacob” in Node 2.

But “Jac” is not in the table.

Go to Jam’s left-most leaf node, which is Node 3
Go to the node’s left sibling, which is Node 2
Search for “Jacob” in Node 2.
A Recap of Wormhole

- Its lookup cost is $O(\log L)$
  - Asymptotically (much) better than $O(\log N)$ and $O(L)$

- Its space cost is comparable to B+ tree
  - One anchor per leaf node

- Efficient support of range Search

- Conduct split/Merge of leaf nodes (for insertion/deletion)
  - Same as B+-tree (But do not need to maintain tree balance)
  - Efficient concurrency control (see the paper)
Performance Evaluation

- **The workload**
  - Amazon reviews metadata (143M keys)
  - URLs from (“http://zwischenruf.at/?p=1012”) (192M keys)
  - Randomly generated keys of different sizes (8B, 16B, ... 1024B)

- **The server**
  - 16-core Intel Xeon E5-2697A, 40MB shared LLC, 256GB DRAM@2400MHz

- **Indexes in comparison**
  - Skiplist
  - B+-tree
  - Masstree
  - Cuckoo Hash Table
Lookup Performance

- At least 1.6x higher
- Up to 5x higher
Compare to Cuckoo Hash Table (w/ Point Search)

ONLY 2x ~ 3x slower
Conclusions

- A new index data structure for sorted keys with an asymptotically low cost of $O(\log L)$.

- A well optimized implementation delivers throughput multiple times higher.

- A promising data structure for fine-grain-indexed big data store.

Source code is available for downloading at https://github.com/wuxb45/wormhole