Benchmarking Ticketing in Vectorized Cuckoo Hash Table for Database Systems

Katherine Yue[1]; Shruthi Kunjur[1]; Ryan Marcus[2]

[1]Penn Undergraduate Mentoring Program, SEAS 2026 [2]University of Pennsylvania, CIS

Background

A hash table is a common computer data structure that stores a collection of key-value pairs, and are regularly used for data retrieval and aggregation, as they can efficiently search, insert, and delete within databases. It uses a hash function to compute an index from a given key into an array of slots, where it places the key and value, and a similar process is performed to find the value for a key in a hash table. However, there are challenges such as collisions, where different keys hash to the same index. A cuckoo hash table is a type of hash table that resolves collisions through an eviction process, ensuring efficient and consistent access times.

Hash tables are key in data science for the common practice of aggregating data, as large amounts of data need to be counted, combined, and used in various formats. In the past, the process of aggregating data involved taking your data, creating a hash table with it, and then iterating row by row based on your desired aggregation process. As part of Penn's new database prototype, FerricDB, we sought to implement a vectorized approach to aggregation, specifically ticketing. Instead of scanning through rows, we scan through columns for the process of assigning distinct "tickets" to keys.

Results

We implemented our cuckoo hash table with optimized and vectorized lookup and insert methods. As the main use case for this implementation doesn't involve deletions from the table, we didn't focus on the delete method. Using our hash table, we implemented a ticketing process, which assigns each row in a table a "ticket". This ticket is essentially an identifier for a key, and once we assign tickets to all the keys in the table, we can identify existing keys, unique keys, and duplicated keys. Overall, this makes aggregation tasks easier and more efficient.

To evaluate our work, we utilized the perf tool for benchmarking, which is a tool that measures the performance of a hash table's methods (time and throughput). For our hash table implementation, our performance with the vectorized insertion of 100,000 elements is shown below.

```
CuckooHashTable/vector insert
                        time:   [3.5505 ms 3.7579 ms 4.0273 ms]
                        thrpt:  [24.830 Melem/s 26.611 Melem/s 28.165 Melem/s]
```

For our ticketing process, we implemented two strategies. One (on the top), as we proceed through the keys chunk by chunk, we first calculate all the hash values for the chunk and then insert them one by one if appropriate. Two (on the bottom), chunk by chunk, we first calculate all the hash values and their known ticket values and then iterate through each key and value to see if a more updated retrieval is necessary. Our performance for both strategies is shown below – note that the "i32 distinct" benchmark processes 1,000,000 distinct i32 values, the "i32 single" benchmark 1,000,000 integers that are the same value.

```
tickets/i32 distinct    time:   [145.56 ms 145.77 ms 146.01 ms]
                        thrpt:  [6.8489 Melem/s 6.8600 Melem/s 6.8699 Melem/s]
tickets/i32 single      time:   [28.816 ms 28.835 ms 28.855 ms]
                        thrpt:  [34.656 Melem/s 34.680 Melem/s 34.702 Melem/s]
```

```
tickets/i32 distinct    time:   [126.18 ms 126.31 ms 126.45 ms]
                        thrpt:  [7.9084 Melem/s 7.9168 Melem/s 7.9252 Melem/s]
tickets/i32 single      time:   [6.9393 ms 6.9416 ms 6.9444 ms]
                        thrpt:  [144.00 Melem/s 144.06 Melem/s 144.11 Melem/s]
```

Comparatively to the original hash table and ticketing implementation, ours was marginally better in some, but not all test cases.

Methodology

We used the Rust language to implement our vectorized cuckoo hash table as well as built-in Linux tools, such as perf, to evaluate the efficiency of our implementation. Furthermore, we integrated our hash table with the FerricDB API, Penn's existing high-performance database prototype.

Conclusion

While there is still a lot of work to be done to improve the efficiency of the cuckoo hash table, we gained many insights from incorporating ticketing and vectorizing numerous processes. After implementing several optimization techniques on our hash table, we were unable to discern a significant improvement in throughput. However, with the improvements that we have been able to make, with further research, we are confident that the efficiency of the hash table can be greatly improved.

Next Steps

Our next steps would involve trying out different strategies for our get and insert functions, ultimately working towards optimizing more and more features of the hash table. We also need to continue to implement vectorization throughout our implementation as well as beyond (which involves other aggregation methods, such as counting and finding the minimum value in a table). There is still much to discover about how vectorization would make a hash table faster, so further research and continued development is necessary.

Acknowledgments

Thank you to Professor Ryan Marcus and the PURM program for offering us the opportunity to be a part of this impactful and stimulating project. We really enjoyed the summer and are incredibly grateful to have gained such valuable research experience through this program!