INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Software Programming for Performance

In-Memory Key Value storage

04.02.2020

Optimizers-2.0:

>> Mohsin Mamoon Hafiz - 2018101029

>> Srivathsan Baskaran - 2018101049

Overview

The project requires us to build an In-Memory Key-Value Storage data structure in C++.
A data structure needs to be built which stores these Key-Value pairs and facilitates the following operations:

1. **get(key):** **Return value for the key**
2. **put(key, value):** **Insert key-value, overwrite value if it already exists.**
3. **delete(key) :** **Delete the key-value from the data structure.**
4. **get(int N):** **Return the Nth key-value pair**
5. **delete(int N):** **Delete the Nth key-value pair from the data structure.**

The project demands these operations on the data structure to be fast, efficient and consuming less memory and cpu while having a high transaction rate.

Ideas

1. The data structure we are planning to use is either AVL Trees or B-Trees
2. We are planning to implement a software cache of size 100 (approximately)
3. Along with the tree data structure, we plan to have a hash table to store the values for keys
4. We are planning to use semaphores to enable multithreaded usage with correct functionality
5. The keys will be stored in the same order as followed by the inbuilt function strcmp()

Description

The explanation for the ideas mentioned is:

Why AVL Trees or B-Trees?

Both AVL Trees and B-Trees provide ordered storage of elements with the time complexities for most common operations like (put, get, delete, etc) as O(log(n)) [n is the number of elements present]. The other possible options were:

1. Binary Search Trees: But B-Tree is a generalized version of BST which provides a much better cache locality.

2. Hash Tables: While get() operation in hash tables is O(1) but the major disadvantage with hash tables is that they are not ordered and hence do not fit the requirements for the project.
3. Tries: The major issue with tries is that they are not memory efficient and in fact use much more memory than AVL Trees or B-Trees.

We researched about a few more data structures but these were the most promising ones and AVL Trees and B-Trees were the best among them with almost similar performance metrics to one another.
We will further test the performance of both AVL trees and B-Trees and select the one which performs with respect to the requirements of the project.

Necessity of a Software Cache:

One of our ideas to solve the problem is to implement a small software cache of size 100. This is proposed in order to optimise fast access to the latest entries inserted. This software cache serves as a fast memory with least access time so that queries on the entries which were added recently could be processed optimally.
As this is a software cache, we are looking forward to implementing a Cyclic Cache eviction policy based on the LRU (Least Recently Used) policy employed on the hardware caches.

Trees + Hashtable?

The main idea behind this decision is that usually in a key-value storage structure, the get(key) operation would mostly be used and while the tree structure would do most of the operations in O(log(n)), the get(key) operations can be optimized to O(1) using a hash table improving the overall performance of the KVStore. Though, here, a trade-off of memory is involved because it will involve twice the memory.
So we are planning to test this rigorously and implement this idea only if the increase in the memory consumption is compensated by the increase in performance.

Synchronizing and Blocking the API calls:

As the project demands the use of Multiple Transaction Threads on the data structure, there is a high risk of the pthreads getting into a Race Condition resulting in a deadlock or a Segmentation Fault.
Proper synchronization among the threads is required for the smooth functioning of all the API calls. This can be achieved by the usage of pthreads and semaphores in the program.
While multiple threads are being scheduled to execute some of the threads must be blocked while others are allowed to execute to prevent deadlocks and seg faults. From the analysis of the given API calls , following are the conclusions derived on blocking of the calls:

API CALL	CALLS TO BE SYNCHRONIZED
Put operation	**put(key,value) , delete(key) and delete(int N)**
Search operations (get(key) and get(int N))	**put(key,value), delete(key) and delete(int N)**
Delete operations	**put(key,value), delete(key) and delete(int N)**

References

I. Introduction to Algorithms 3rd Edition (Charles E. Leiserson, Thomas H. Cormen, Ronald L. Rivest)

II. https://www.geeksforgeeks.com

III. https://arxiv.org/pdf/1907.01631.pdf