Implementation specification

Yoogottam Khandelwal 2018101019
Bhavyajeet Singh 2018111022

Requirements:

- An In-Memory Key-Value Storage Software in C++
- Supported APIs
  - `get(key)` : returns value for the key
  - `put(key, value)` : add key-value, overwrite existing value
  - `delete(key)`
  - `get(int N)` : returns Nth key-value pair
  - `delete(int N)` : delete Nth key-value pair
- Spec
  - Max key size: 64 bytes
  - Each key char can be (a-z) or (A-Z): Total 52 possible chars
  - Max Value Size: 256 bytes, any ASCII value
  - Keys according to `strcmp`

Class definition

```
struct Slice{
    uint8_t size;
    char*   data;
};

class kvStore {
  public:
    kvStore(uint64_t max_entries);
    bool get(Slice &key, Slice &value): //returns false if key didn’t exist
    bool put(Slice &key, Slice &value): //returns true if value overwritten
    bool del(Slice &key);
    bool get(int N, Slice &key, Slice &value): //returns Nth key-value pair
    bool del(int N): //delete Nth key-value pair
};
```

Evaluation benchmark

- Multithreaded benchmark application
- Runs:
    - Benchmark would first load data via put calls (10 million entries, 2 min time limit)
    - Perform Single Threaded Transactions to verify kvStore functionality
    - Multiple Transaction Threads with each thread calling one of the APIs

---

Data structure

We have a few data structures in mind. We will try them all and check which one gives the best overall result.

- Self balancing BST (RB tree) / B+ Tree w Hash table
- Trie (compressed trie) w Hash table
- Finite State Transducer (FST)

Discussion

Red-black tree

A BST is very simple. It can satisfy all requirements. I think this will be a good place to begin. `strcmp` will tell us which branch to follow. For getting the n^{th} item, we will augment the data structure by adding number of nodes under that node for every node.

The problem with BST is that it is going to take a lot of space. Each node in the BST will require 64 bytes for storing the key. 256 bytes for storing the value. This brings the total to 320 * 10^7, which is a little over 3.2GB. Each `strcmp` will have to go through the whole key.

Using a self-balancing BST like red-black tree will ensure that insertions/deletions/updates are fast enough (`O(log(N)) amortized`). We are not using AVL tree since it is a lot stricter compared to RB Tree and it would waste a lot of time in rotating to make it balanced.

B+ Tree

A B+ tree is also a search tree. It is not restricted to two children per node. We can decide the branching factor by trying out different powers of 2.

The space requirement is almost the same but the complexity of implementing it is higher than that of red black tree.

B-trees and it's families are used in databases for minimizing disk reads and writes. This is why we thought that it might be useful. We are not sure whether it will actually be beneficial for us.

Trie

We are planning to use a compressed trie. The space requirement for trie will be much lesser since we can combine nodes for common substrings. The value corresponding to each key will be stored at the leaf.

For storing the 'next' pointer (pointer to the next character), instead of maintaining a list for all 52 characters, we can store a small BST (balanced or otherwise, since only 52 elements can be there). If we do this, we will save a lot of space since most of it will go empty.

For getting the n^{th} element, we can store number of keys ending under each node.

FST

FST uses the finite state machine as a data structure in order to store key value pairs. It was designed specifically for the purpose of storing ordered key value pairs and accessing them.

Using an FST would increase the transactions per second with a huge margin but the implementation complexities may not allow achieving it in its complete form within due course of time

Cache optimizations

In case if we choose to not use tries, along with optimizing the hardware cache optimizations, we can use a hash table which stores some of the key-value pairs to be able to answer `get()` API calls quickly.

This would act as a software cache which would be updated every time a new entry is inserted. This updation policy has been chosen since the newer entries would probably be less likely to be accessed compared to the old ones.

The reason to not implement a hash table in case of usage of tries is that in order to hash you must go through the entire string length which involves the same computation as going through the trie itself.

Multithreading

- We would be using the pthreads C library in order to implement parallelism
- Since every function would be atomic and there is not much scope to parallelism the functions, the parallelism would come in by handling multiple requests simultaneously
- In order to do so we must make sure that the consistency and integrity of the data is maintained and hence the principles of reader-writer's problem would be implemented
- to avoid the over-head of thread creation each time, we plan onto having a limited number of threads created beforehand, like a threadpool

- whenever a thread is trying to write or delete, other threads must wait, whereas multiple threads can read simultaneously

Memory Managment

The Major objectives while deciding the memory schema of the program involved **minimising the maximum memory usage** and also at the same time maximising the transactions that take per second

To do so the following methods have been chosen

- To avoid repetitive malloc calls, large chunks of memory would be allocated and then filled up in due course
- as soon as one chunk of memory is entirely consumed (with howsoever number of entires), another chunk of same size would be allocated
- Each delete operation would not result in freeing the memory space, instead a separate list of available memory spaces would be implemented
- The separate list will contain the size and pointer to every piece of available space
- whenever a new data entry is to be inserted we find the best size fit for that new entry from our list and place the entry in that segment
- This would result in least memory wastage and at the same time would reduce the overhead of many malloc and free calls

Other optimizations

We will apply bit hacks wherever possible. It wasn't clear where these can be applied without actually implementing the data structure first.

References

- https://blog.burntsushi.net/transducers/
    - http://www.researchgate.net/profile/Jii_Dvorsky/publication/221568039_Word_Random_Access_Compression/links/0c96052c095630d5b3000000.pdf#page=116
    - http://www.cs.put.poznan.pl/dweiss/site/publications/download/fsacomp.pdf
    - http://www.mitpressjournals.org/doi/pdfplus/10.1162/089120100561601
    - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.3698&rep=rep1&type=pdf
- https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures.html
- https://en.wikipedia.org/wiki/List_of_data_structures#Trees