PROJECT REPORT

IN-MEMORY KEY-VALUE STORE

Software Programming for Performance

SpacesOverTabs

Bharathi Ramana Joshi 2019121006

Gaurang Tandon 2018101091

Jivitesh Jain 2018101092

Tanmay Sachan 2018111023

Contents

Problem Statement Analysis

Definition

The problem statement requires us to build an *in-memory key-value storage software* with the following specifications:

- Supported APIs:
  - `get(key)`: get the value of the key
  - `put(key, value)`: add or overwrite the key-value pair
  - `del(key)`: delete the key and its associated value
  - `get(int n)`: get the n^{th} key-value pair
  - `del(int n)`: delete the n^{th} key-value pair
- Keys are character arrays containing *A-Z* or *a-z* characters, up to 64 bytes in length
- Values are character arrays containing ASCII values, up to 256 bytes in length
- Keys should be ordered according to the ordering followed by the `strcmp` function defined in the header file `cstring.h`
- API calls should be blocking
- Transactions per seconds should be as high as possible, while the average CPU usage and peak memory usage should be minimised
- The system should be able to handle up to 10^7 entries

A key-value store, dictionary or hash-map is an associative array, that is, it stores a collection of key-value pairs, with each possible key appearing at most once in the collection. The lack of a predefined structure in the data makes key-value stores a type of NoSQL databases, that is, databases that do not follow the relational data model. This particular implementation has to be in-memory, meaning that it cannot write to the disk and has to support several threads calling the APIs simultaneously.

Challenges

Memory Requirements

Each key is, at the longest, a 64-byte character array consisting of *a-z* or *A-Z* characters. Thus, it contains 64 characters (as each character takes a byte). However, because only 52 characters are allowed, each can be represented in just 6 bits instead of 8 (as $2^6 = 64 > 52$). Hence, each key requires 64 × 6 bits or just 48 bytes of memory.

Each value is, at the longest, a 256-byte character array consisting of any ASCII value. Because ASCII values range from 0 to 255, no compression is possible here.

Thus, it takes 256 + 48 = 304 bytes to store the data pertaining to a key-value pair. Summed up over 10^7 entries, this accounts to around **2.83 gigabytes**, under the assumption that the key was actually compressed.

However, this calculation does not take into account the pointers, indices and other metadata that needs to be stored to operate the data structures that hold this data. Hence, the memory requirements can be easily expected to double to almost **5 gigabytes**.

Data Structures

Binary Search Trees

Balanced binary search trees are commonly used to implement set and map data structures, such as those in C++'s STL, hence they were an alternative we initially explored - as stated in our implementation specification.

They work by storing key-value pairs in the nodes, with lexicographically smaller keys appearing in the left subtree and larger ones in the right. A key lookup involves comparing the key at hand with the one stored in the current node and moving to the left or right subtree as per their lexicographical ordering. The

values can be retrieved in sorted order via an inorder traversal of the tree. Retrieving the N^{th} key-value pair would require augmenting the data structure by storing in each node the size of its subtree.

Balance is maintained by using a colour bit (and maintaining certain invariants) in case of a red-black tree, storing more than one key per child in case of b-trees, inducing rotations in case of AVL trees etc. This keeps the height of the tree capped at *O(log(number of entries))*.

However, the comparisons require an *O(key length)* compare operation at each level in the tree, raising the search, insert and delete complexity to *O(tree height × key length) = O(log(number of entries) × key length)*. Hence, although one of our first working implementations was a red-black tree, **this prohibitively large complexity later made us decide against using binary search trees.**

Deterministic Acyclic Finite State Transducer

We had explored the prospect of a Deterministic Acyclic Finite State Transducer in our implementation specification. It is basically a trie, with the various branches converging again towards the end if they share common suffixes. Thus, it allows for insert, lookup and search operations to be carried out in *O(key length)*, while using less memory than a trie.

However, several problems plague this data structure. It is usually used for handling integer data types, and its implementation for non-integer ones is especially complex. It also requires an auxiliary hash table in order to detect and merge equivalent suffixes to keep the memory requirements low. However, interleaved accesses to the hash table and the main data structure lead to poor spatial locality and an increased cache miss rate.

This made us decide against using this data structure as well.

Van Emde Boas Trees

A vEB tree is a data structure which supports the operations of an associative array and works by recursively splitting a universe of size n into √n universes of size √n at each level.

While we had discussed this data structure in detail in our implementation specification, we had also talked about its problems, such as its requirements of integer keys (which would require a hashing function, and hence would cause a loss of ordering information). **Hence, we decided against using this data structure as well.**

Tries

Tries are a time-tested data structure for sorting and searching strings. We can use tries to implement a key-value store by storing the keys along the path and the values at the final nodes. Thus any insert, search or delete operation requires traversing down the trie at most up to a depth equal to the length of the key, resulting in a complexity of *O(key length)* for these operations.

Tries can easily be modified to store the number of leaves in each node to support `get(int n)` and `del(int n)` operations.

Thus, **we decided to focus on tries.** Our second implementation was purely trie-based and supported all the required operations. However, the memory usage of tries is large, and we had to move towards compressed tries for keeping our memory requirements low.

Compressed Tries

Compressed tries improve upon tries by storing common substrings in a single node, for as long as no splits are required. This significantly reduces the memory required, as well as the search times. Thus, they fit our use case perfectly, and **our final implementation uses a compressed trie.**

Memory Allocation

We had initially decided to implement a memory-allocator and manage memory ourselves, avoiding library calls to `malloc`.

However, we realised that performance gains from this were not significant, and the implementation was only adding to the complexity of the system. Moreover, `malloc` internally manages and reuses allocated memory, keeping system-calls to `mmap`, which actually requests the OS for memory (and is hence slow) to a minimum.

This suits our needs well and we have simply used the C++ keywords `new` and `delete` for memory allocation. However, we have paid special attention to minimising memory leaks, and have written **custom constructors, destructors and move constructors** for several of our classes.

Concurrency

As per the requirements, the key-value store should support concurrent calls to its interface. We have taken care of this requirement, and the headaches that accompany multithreaded access to shared memory by using **mutex locks and semaphores**. We have implemented a standard **reader-writer system**, wherein multiple threads can read the memory at once, but only one can write. This allows us to maintain correctness, whilst keeping bottlenecks to a minimum.

Multithreading

While we initially considered parallelising internal operations as well, we realised that most of the operations that could be parallelised (such as searching through the list of children of a trie node) were relatively small (a trie node can hove only 52 children) and thus the overhead of creating a thread could not be justified. **Thus, our implementation does not use multiple threads internally.**

Other Optimisations

Binary Search Tree inside a Trie Node

Each trie node maintains a list of its children, which is conventionally done using an array as big as the size of the alphabet, which in our case is 52. While this array allows for constant-time access to the elements, it wastes memory because not every node has all 52 children.

Thus, we augmented each trie node with a custom BST to store its children, wherein each node is created only when it is required. The extra time for searching the BST is a mere *O(log(52)),* and thus insignificant. This allowed us to significantly reduce our memory requirements.

Branch Reductions

Wherever possible, we tried to reduce branches and `if` conditions by using tricks relating to, for example, the fact that C++ stores `false` as `0` and `true` as `1`. We have also used `unique pointers`, which are smart pointers that prevent memory mismanagement and allow for compiler optimisations. Several such small optimisations are scattered throughout our code.