Efficient Key-Value Storage in C++

Team <Placeholder_name>

18 February 2020

Arpan Dasgupta (2018111010), Bhavyajeet Singh (2018111022), Jaidev Shriram (2018101012), Yoogottam Khandelwal (2018101019)

Abstract

This is a project for the course Software Programming for Performance at the International Institute of Information Technology, Hyderabad. The project implements an in-memory key-value storage system. The encapsulated methods and data structure have been optimized to handle millions of requests while keeping memory and CPU usage to a minimum. This report explores various data structures and their pros and cons. Additionally, it explores other optimizations and implementation details relevant to the storage system.

Contents

1 Introduction

This reports builds on the previously submitted implementation specification document for the project. The project implements an in-memory key-value storage system. The encapsulated methods and data structure have been optimized to handle millions of requests while keeping memory and CPU usage to a minimum. This report explores various data structures and their pros and cons. Additionally, it explores other optimizations and implementation details relevant to the storage system. This document is meant to be a reflection about the work that has been put in along with various design decisions that went into the project. Hence, the gist of discussions that took place when considering certain parameters have been included.

2 Progress after Previous Implementation Spec

In the previous Implementation spec we planned to test out out various data structures like red black tree, trie, and B-tree. upon further analysis we narrowed down to testing and implementing tries and b trees. FST was removed out of consideration since the implementation highly complex. Implementation and detailed analysis of B tree and trie was done inn the given timespan and the conclusions have been mentioned below. In aspects of memory, for now we malloc for every node that is created but later on we plan to optimise that by reduce the amount of malloc calls our program does. // When it comes to implementing concurrency, for the basic minimal working code, we did account for any multithreading, neither from the benchmark's perspective nor from our code's perspective but we plan to do so for the final submission of the program.

3 Progress after First Submission

We have made significant gains in performance over the past week as the compressed trie implementation worked. This dramatically reduced the memory usage while simultaneously giving us the performance benefits of the trie.

The static memory allocation strategy suggested in the previous report has also been implemented successfully.

4 Binary Search Tree

BST was considered as an option for the given task because it can be used to store ordered key value pairs and would perform get,delete and insert operations at an average O(log(n)) time.

4.1 Using regular BST

For regular BST we do not perform any additional optimizations or self balancing for the tree and let the key value pairs be added to the tree in the same order as they are inserted. Each node represents a key and the nodes are arranged in the lexicographical order i.e. for any node, if the the key that we are looking for is smaller in lexicographical order we go left, otherwise to the right.

Parallelism can also be implemented in regular bst by allowing multiple reads simultaneously and for every write or update we only lock the entire subtree of the node that we are currently traversing. Ultimately a BST was discarded due to the uncertainties involved, and if the order of insertion is not a favourable one then even very few entries can increase the tree height significantly.

4.2 Using balanced BST

A balanced BST would provide additional efficiency over a regular BST since there is a certainly that the maximum height of the tree would not cross log(n) where n is the total number of entries.Though along with that there would be a very minimal overhead of keeping the tree balanced after every insertion/deletion. various implementations of a balanced BST like AVL, red-black etc were considered but they were discarded because the overhead of rotating the AVL tree was significantly higher.

4.3 B-Tree Performance

A B tree is a further modification to the basic BST, but here the number of children a node can possess is not restricted to just 2, on top of that, a B-tree is a self balancing tree and thus all the leaf nodes of a B-tree are necessarily at the same level at every point. a B-tree would provide extra optimisations over a BST or a balanced BST since the height of the tree would always be O($log(n)$).

B-tree of also chosen because there is further scope of optimising it by finding

the optimum order of the tree and implementing parallelism/concurrency.

4.3.1 N-order B-Tree

The order of the b-tree is given by the maximum number of values a particular node of a B-tree can hold. As we increase the order the height of the tree decreases but at the name time the number of linear comparisons required at each node increase. Furthermore with increasing order the total memory consumption also decreases since the total number of node decreases. And since every node corresponds to a pointer, the total number of pointers stored decreases.

4.3.2 Advantages

1. There is no randomness involved since the height of the tree is always going to be at max O($log(n)$)
2. There is scope of further optimization since we can find the best value of the order depending upon the maximum entries
3. Parallelism can be implemented by allowing multiple simultaneous reads and implementing locks on the subtree of the current node while traversing for an update or delete
4. it outperforms other data structures like Trie in terms of memory performance

4.3.3 Disadvantages

1. Transactions per second are less compared to a trie
2. Ordered requests i.e. get(n) or delete (n) take longer since in order to get the ordered output in a naive BST you must traverse through all entries

5 Trie

A trie is an optimal data structure for strings considering the constant time for insertion, deletion and searching. This performance is a trade-off with memory however. We consider trie because of the massive performance boost that is expected with such a data structure. The benchmark score used to grade the product prioritizes Transactions per Second (TPS). Hence, serving multiple

requests within a certain time frame is more than ideal. Further, using a trie would also permit some degree of concurrency in writes and reads as the entire data structure needn't be locked to perform these operations.

This parallelism is permitted by the property that the path to a key-value pair remains unchanged through the life of a trie. While, in a B-tree, due to the constant rearrangement of nodes, more than one node must be locked.

5.1 Regular Trie

A regular trie (or what this report will consider one as) is characterized by the following properties:

1. Each node in the trie is a single character
2. Depth of the trie is the maximum length of all strings
3. Each node in the trie has an array which can point to any other character

Each of the above points proved to make using a regular trie expensive memory wise. For just a single string of at max 64 characters, a trie of depth 64 is created. This means that at every node, there are 51 other nodes which point to NULL but exist regardless. This is a massive waste of memory clearly. But, the TPS is very high for this data structure.

5.2 Trie with linked list

One problem with trie was the memory wastage due to NULL values in the array of pointers at each node. A solution to this was removing an array and using a linked list which store the letter and the pointer to the node. Obviously, this would be unnecessary if a node has 52 pointers, but considering that it is an unlikely event, linked list was expected to be an improvement.

After implementing this version, we discovered that it was in fact slower while saving memory. The reduction in speed is undoubtedly because of the linear time in moving from one node to another as a result of linked list traversal. Our calculations indicated that nodes should be sparse at higher depths but the real time results proved that other factors voided any benefit at larger depths. Hence, trie with linked list was abandoned.

5.3 Trie with BST

Building on the same idea proposed earlier, a BST at each node to navigate to the required letter should give a performance improvement. Here too, we store just the pointers that actually point to a letter rather than having an array of 52. It is to be noted, that the cost of performance with a BST is additional memory. For every letter in our node, our BST will have 2 other pointers for the letter. Hence, when one third of 52 letters is used, we end up using as much of space as the array. However, as we increase the depth, the chances of multiple letters in a node decreases. Hence, we felt the need to implement this and see performance on millions of random strings.

Surprisingly, this was better than the regular trie and the trie with linked list performance wise and memory wise. The degree of improvement wasn't significant but it was still noticeable.

6 Transition to Compressed Trie

The previous three implementations made one thing clear - the performance of trie is unbeatable but the memory usage makes it very expensive. In an attempt to retain the performance benefits of trie, we determined some key aspects of the application that must be made:

1. The strings vary in length significantly - from several strings of length in the lower 10's to strings near 64, there is quite a bit of variance. Hence, even if 10 million entries exist in the trie, the depth of the trie for various words varies significantly (significant has been used quite often in this context because of the memory used per node, not because of the number of nodes alone).
2. There is no character specific information to be stored - Hence, words needn't be broken down into individual characters.
3. Memory usage must be dynamic - explored in greater detail later. Since memory allocation is a costly process, we want to minimize the number of times this happens as well.

The above factors lead to the idea of compressing the trie. By compression, we mean that no new nodes are created for words when there is no other word that follows the same path.

We propose the following standard idea of compression for a trie to group the suffix of a word and split the suffix further when a word is being inserted which clashes with the first letter of the suffix.

6.1 Previous Implementation

For example: Consider the words "abc", "bca", "cbaaaaa". In a regular trie, this would be a tree of depth 3 - one level for each word. In a compressed trie, this would be a tree of depth 1.

This is because the suffixes 'bc', 'ca', and 'baaaaa' can be stored as suffixes to the letters 'a', 'b', and 'c' respectively. Since there are no other words which share a prefix with our suffix, it is guaranteed that this is a safe representation of our data.

Let's take another example where there is a prefix that clashes:

The words being inserted into the trie are: "abc", "aardvark", "bc", "aid".

Now, it is clear that we cannot have a tree of depth one for this trie as we have three words that start with the letter 'a'. However, none of these share a prefix with our remaining word. Hence, at the second level of the trie, we can group 'bc', 'ardvark', and 'id' together with the first letter of each being the node at which the suffix is attached.

When we are inserting a new word into this, such as 'abd', we face another problem. The suffix 'bc', and 'bd' share a common prefix 'b'. Hence, the former suffix is further broken down until it no longer shares a prefix with the latter. This will increase the depth of the trie.

It is this suffix storage that gives a considerable memory improvement. By making massive gains in the depth of the trie, we are reducing the number of memory allocation calls apart from reducing wastage of memory. There is slight overhead in breaking a suffix (such as the previous example) but this is negligible.

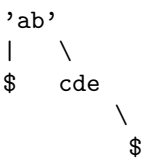
Our team has implemented a rudimentary compressed trie which was the best performer among all data structures used so far. When tested on a 4th gen intel i7 PC, the trie is capable of inserting $10\hat{6}$ elements in just under 3 seconds, using 1.5GB. This is a considerably improvement from the naive trie that was implemented earlier.

The only data structure that performed better in terms of memory usage was the B-Tree.

6.2 Current Implementation

The current compressed trie can be concisely described as a trie which compressed prefix, and suffix to the maximum extent possible.

For instance, for the words 'ab', and 'abcde', the structure of the trie would be:



This dramatically reduces the height of the trie once again. This was undoubtedly challenging to implement and is a hop away from a FST, which is great for performance. Memory wise, this trie didn't perform too differently from the previous compressed trie but the time for insertion of $10^6 keys on our test PC was under 1 second - an 8 fold improvement over the B-Tree$!

6.3 Concerns with Compressed Trie

The submitted at the code at the moment does not use compressed trie. Unfortunately, there are some memory leakage concerns with this particular implementation, including writes to regions beyond memory. While these haven't proven to be fatal so far, it is unpredictable and rarely occurs. Until we can confidently implement a version free of these flaws, it is not safe to use the compressed trie.

7 Concurrency Concerns

In order to improve performance and to make the best out of our resources, we plan to implement concurrency by using the c pthreads library. But in order to do so we must ensure the correctness and consistency of the program and the ability to tackle the race conditions that come in hand with parallelism. Since most machines that we use today have multiple cores, a concurrent program would perform much better than a single threaded one.

7.1 Locking

Locking of shared memory space is necessary to ensure the correctness of the program. A lock ensures that only a single thread can access the locked memory region and all other threads that wish to do so until the lock is released. This is necessary to avoid race conditions and maintain the consistency of the database.

7.1.1 Mutual Exclusion

Complete mutual exclusion was avoided because it is not necessary to lock the entire storage every time when someone is trying to access it. Instead the lock must be implemented only when a thread tries to update (insert or delete values) into the key-value store because in that case the structure and lexicographical ordering of the storage is likely whereas we can allow multiple simultaneous reads since a read (get operation) does not change anything is the structure.

7.1.2 Reader Writer Lock

A Reader Writer Lock refers to the implementation of the above mention concept where a reader does not lock the storage and thus leaves it open for other readers to access but a writer ensures exclusivity and does not allow any other reader or writer to access the storage. If there is an ongoing read operation going then the writer must wait until the operation is over. For this we plan to use the inbuilt reader writer locks from the c++ pthread library.

7.1.3 Implementation and Overhead Realizations

When locks were implemented in fine detail in the trie - locks at every node, we noticed a 10 fold increase in runtime for the same function! This can only be attributed to the overhead of obtaining the lock.

When a single lock was used for the entire database, the runtime was significantly lower. This is undoubtedly contrary to expectation as concurrency is reduced, but the overhead of locking is much more than the actual runtime of the function, which makes this approach impractical.

8 Optimizations

There is immense scope for further optimizing the current code by applying various optimisation techniques learnt throughout the course but the two most significant optimisations which are to be done are as follows:

8.1 Static Memory Consideration

Since each TrieNode and Slice structure is of constant size, we can statically allocate this initially itself using a free list. This is essentially a linked list which contains pre-initialized nodes. As long as the head of this list, is not null, we can just get a node from this list and move the head to the next node, rather than doing a malloc.

This has proven to be a significant memory booster, saving about 2 seconds for insertion of around $5*10^6 numbers. However, memory allocations for the variable data in a Slice can't be allocat$
$255, and the memory wastage on average would be significant.$

Additionally, we considered allocating a fixed block size to the data and then performing a realloc operation to get extra memory if required. However, the overhead of realloc, combined with the persisting problem of internal fragmentation made this approach unnecessary.

8.2 Multi-threading Functions

Since the functions (requests) are atomic there is not much scope to parallelise the functions but the parallelism would instead come from handling multiple request simultaneously. This would increase the transactions per second significantly and would also increase the total CPU usage

9 Observations

9.1 Inaccuracy of Testing

The reader must note that the testing environment varied from time to time and the results for timing and memory usage is dependent on other programs running on the system. Further, the performance depends on the underlying hardware, which makes predictions purely based on the test PC. Sometimes,

due to RAM limitations, swap space was used which further slowed down the PC.

Hence, all observations here must be taken with a grain of salt. However, variations in performance shouldn't be too different. At the very least, the comparative performance will remain the same from PC to PC.

10 Conclusion

The code submitted contains implementations through B-tree and compressed trie data structures. For the current submission only the correctness for a single threaded benchmark has been ensured along with a detailed analysis of memory and time usage for various inputs. We plan to further optimise the program by using the methods and details specified above.