Software Programming for Performance

Implementing Key Value store in C++

Project Report

Team - {dict}ators :
Santosh Mylaram (2018101030)
Vinay Kumar Tadepalli (2018101048)
Sai Shashank Kalakonda (2018111016)
Sachin Kumar Danisetty (2018101108)

Implementation Specs:

TRIE :

Trie is a tree-like structure whose nodes store the letters of an alphabet.

We store keys and associated values as an entry in Trie. The key is always a string, but the value could be of any type, as the trie just stores.

So tries can be a better option for this project as key given will always be a string and the arrangement of keys in lexicographical order will be done by the trie itself as it is one of the properties of the trie. To get a lexicographical order, we just need to do a preorder traversal.

Each node in the trie has 52 children and the height of the trie would be 64. Each path from the root to the node represents a string i.e; a key and the node stores the corresponding value of the key.

Each node stores the following information:

1. Value - Stores the corresponding value of the key and size of the key in Slice format.
2. no_of_ends - Maintains the count of no of leaves in the subtree with the current node as root.
3. isEndOfWord - To determine the end of word.
4. TrieNode *child[53] - Contains the pointers to its 52 children and it's parent.

Each node represents a key obtained by the concatenation of all the characters of its Ancestors.

Implementing Functionalities:

1. **put(key,val) :**

   Traverses through the trie to put the value to the corresponding key.If the key already exists it replaces the previous value and returns true. If the key doesn't exist, It creates a new node with the corresponding value to the key. It also updates the variable **no_of_ends** in each of the ancestor of the new node inserted(If a new value is being put)

   The complexity of this functionality is O(h) where h is the height of the tree. Since in the worst case, we have to traverse from the root to the leaf.

```
bool put(Slice &key, Slice &value)
        {
            pthread_mutex_lock(&m_lock);
            bool isThere=true;
            struct TrieNode *curNode = root;
```

```
            for (int i = 0; i < key.size; i++)
            {
                int index = key.data[i]-65 - (6 & ( key.data[i] >=
'a'));


                  if (!curNode->children[index])
                {
                    curNode->children[index] = getNode();
                    curNode->children[index]->children[52] = curNode;
                    isThere=false;
                }
                curNode->children[index]->no_of_ends+=1;
                curNode = curNode->children[index];
            }
            bool isEnd=curNode->isEndOfWord;
            Slice *val;
            val=(struct Slice *)malloc(sizeof(Slice));
            val->size=value.size;
            val->data=value.data;
            curNode->isEndOfWord = true;
            curNode->value=val;
            size++;
            if(isEnd)
```

```
        {
            size--;
            for (int i = key.size-1; i>=0; i--)
            {
                curNode->no_of_ends-=1;
                curNode = curNode->children[52];
            }
        }
        pthread_mutex_unlock(&m_lock);
        return isThere && isEnd;
    }
```

2. **get(key, &val):**

   Traverses through the trie to find the key and assigns the corresponding value to val and returns true if the key exists. Otherwise returns false.

   The complexity of this functionality is O(h) where h is the height of the tree. Because in the worst case, we have to traverse from the root to the leaf.

```
    bool get(Slice &key,Slice &value)
    {
        struct TrieNode *curNode = root;
```

```
            for (int i = 0; i<key.size; i++)
            {
                int index = key.data[i]-65 - (6 &
-(key.data[i]>='a'));

                if (!curNode->children[index])
                {
                    return false;
                }
                curNode = curNode->children[index];
            }
            if(curNode->value!=NULL)
                value=*curNode->value;
            return (curNode != NULL && curNode->isEndOfWord);
        }
```

3. **del(key) :**

   Traverses through the trie to find the key and deletes the corresponding node in the trie and frees the corresponding memory. It also updates the variable **no_of_ends** for all the nodes which are the ancestors of the deleted node(If the node is deleted).

The complexity of this functionality is O(h) where h is the height of the tree. Because in the worst case, we have to traverse from the root to the leaf.

```
        bool del(Slice &key)
        {
            pthread_mutex_lock(&m_lock);
            struct TrieNode *curNode = root;
            for (int i = 0; i<key.size-1; i++)
            {
                int index = key.data[i]-65 - (6 &
-(key.data[i]>='a'));

                if (!curNode->children[index])
                {
                    pthread_mutex_unlock(&m_lock);
                    return false;
                }
                curNode = curNode->children[index];
            }

            int i=key.size-1;
            int index = key.data[i]-65 - (6 & -(key.data[i]>='a'));

            if(curNode->children[index]==NULL)
            {
                pthread_mutex_unlock(&m_lock);
```

```
            return false;
        }
        size--;
        struct TrieNode *temp=curNode;
        temp=temp->children[index];
        for (int i = key.size-1; i>=0; i--)
        {
            temp->no_of_ends-=1;

            temp = temp->children[52];
        }
        if(curNode->children[index]->no_of_ends==0 &&
curNode->children[index]->isEndOfWord)
        {

            free(curNode->children[index]);
            curNode->children[index] = NULL;
        }
        else
        {
            curNode->children[index]->value=NULL;
            curNode->children[index]->isEndOfWord=false;
        }
        pthread_mutex_unlock(&m_lock);
        return true;
    }
```

4. **get(N, &key, &val):**

Traverses through the trie to find the Nth smallest key in Lexicographical order and assigns the corresponding key and value to the pointers key and val. The normal time complexity of this would be O(n), where n is the total no of nodes in the trie, Because in the worst case we have to iterate over all the elements to find the Nth smallest string. But this complexity can be reduced as discussed in the next section.

```
bool get(int N, Slice &key, Slice &value)
{
    if(size<=0 || size<=N)
        return false;

    pthread_mutex_lock(&m_lock);
    N++;
    string key_here="";
    struct TrieNode *curNode = root;
    while(!(N==0 && curNode->isEndOfWord))
    {
        for(int i=0;i<52 ;i++)
        {
            if(curNode->children[i])
            {
```

```
                    if(N-curNode->children[i]->no_of_ends <=
0)

                    {

                        if(curNode->children[i]->isEndOfWord){

                            N--;

                        }

                        curNode=curNode->children[i];

                        if(i>=26)

                        {

                            key_here+=i+'a'-26;

                        }

                        else

                        {

                            key_here+=i+'A';

                        }

                        break;

                    }

                    N-=curNode->children[i]->no_of_ends;

                }

            }

        }


        Slice key1,value1;
```

```
        char *k=(char *)malloc(key_here.length()+1);
        copy(key_here.begin(),key_here.end(),k);
        k[key_here.length()]='\0';
        key1.data=k;
        key1.size=key_here.length();
        value1=*curNode->value;
        key=key1;
        value=value1;


        pthread_mutex_unlock(&m_lock);


        return true;        //assuming given N in range
    }
```

5. **del(N) :**

   Traverses through the trie to find the Nth smallest key in Lexicographical order and deletes the corresponding key-value pair,i.e; the corresponding node in the trie. It also updates the variable **no_of_ends** for all the nodes which are the ancestors of the deleted node(If the node is deleted).

   The normal time complexity of this would be O(n), where n is the total no. of nodes in the trie, Because in the worst case we have to iterate over all the

elements to find the Nth smallest string. But this complexity can be reduced as discussed in the next section...

```
bool del(int N)
{
    if(size<=0 || size<=N)
        return false;
    pthread_mutex_lock(&m_lock);
    N++;
    struct TrieNode *curNode = root;
    int i;
    while(!(N==0 && curNode->isEndOfWord))
    {
        for(i=0;i<52 ;i++)
        {
            if(curNode->children[i])
            {
                if(N-curNode->children[i]->no_of_ends <= 0)
                {
                    if(curNode->children[i]->isEndOfWord){
                        N--;
                    }
                    curNode=curNode->children[i];
                    break;
                }
                N-=curNode->children[i]->no_of_ends;
```

```
                }
            }
        }
        struct TrieNode *temp=curNode;
        for (int i =0; i>=0; i++)
        {
            temp->no_of_ends-=1;
            temp = temp->children[52];
            if(temp==root)
                break;
        }
        if(curNode->no_of_ends==0 && curNode->isEndOfWord)
        {
            free(curNode);
            curNode->children[52]->children[i] = NULL;
        }
        else
        {
            curNode->value=NULL;
            curNode->isEndOfWord=false;
        }
        size--;
        pthread_mutex_unlock(&m_lock);
        return true;
    }
```

Optimizations Performed:

Optimizing Lexicographic searches :

As we are maintaining the leaf count for each node, we can directly skip some subtrees and directly search the subtree in which the Nth smallest key is located. So, the complexity of these operations can be reduced from O(n) to O(h*52), where n is the total no of keys and h is the height of the tree and 52 corresponds to the number of children.

Bithacks:

We used bithacks for computing the index of the child using the character in the string.

Function Inlining:

Inline functions are used wherever necessary like computing indices.

Loop Unrolling:

We tried loop unrolling but the outcomes were not favourable so we omitted them.

Selection of **TRIE** :

The Complexity of most of the functionalities is O(h) where h is the height of the tree and the maximum value of h is 64 as the maximum length of the key is 64. So, each is performed in constant time.

Memory usage is optimized, as the nodes of the trie are created only on the call of **“put()”.**

We didn’t choose hashing because the search complexity for Nth key-value raises up to O(n) where n is the total no.of keys present. And also to perform put(), get() and del() functions in constant time, all keys have to be hashed in the upfront which leads to the wastage of memory as the whole memory required needs to be allocated at start itself.

NOTE :

Multi-Threading :

Since the benchmark code is using pthread library for multi-threading, We are using **MUTEX LOCK** to prevent deadlocks, race conditions, etc.

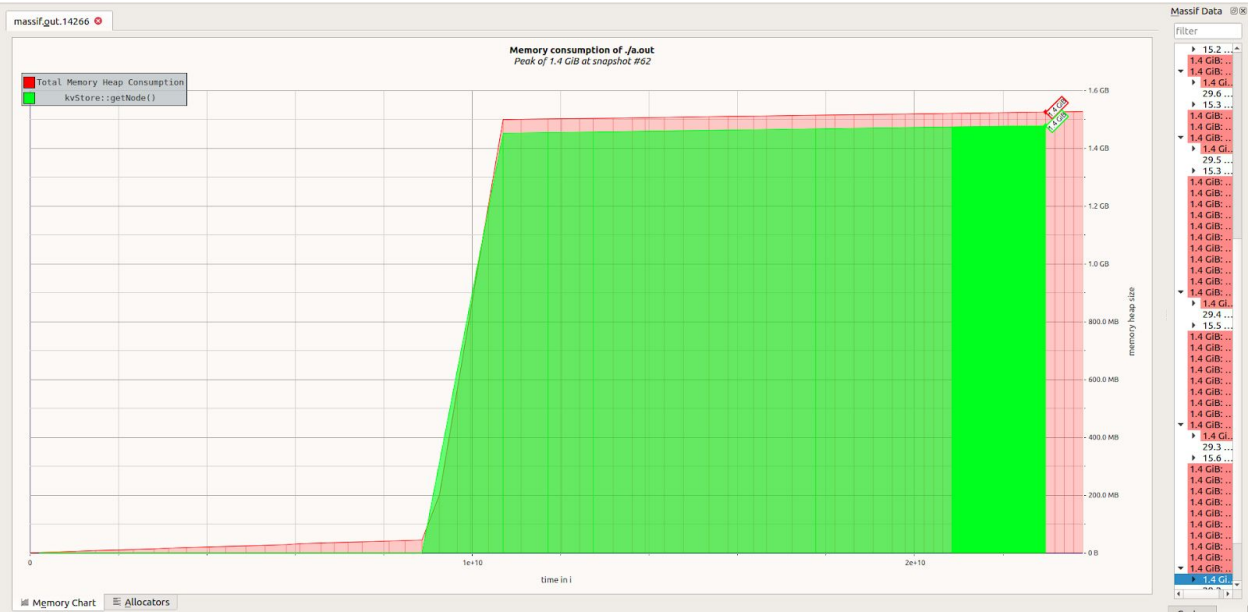
The mutex lock is named **m_lock.**

OBSERVATIONS :

Massif :

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

The observation graph shows that peak memory usage is **1.4GB.**



Perf :

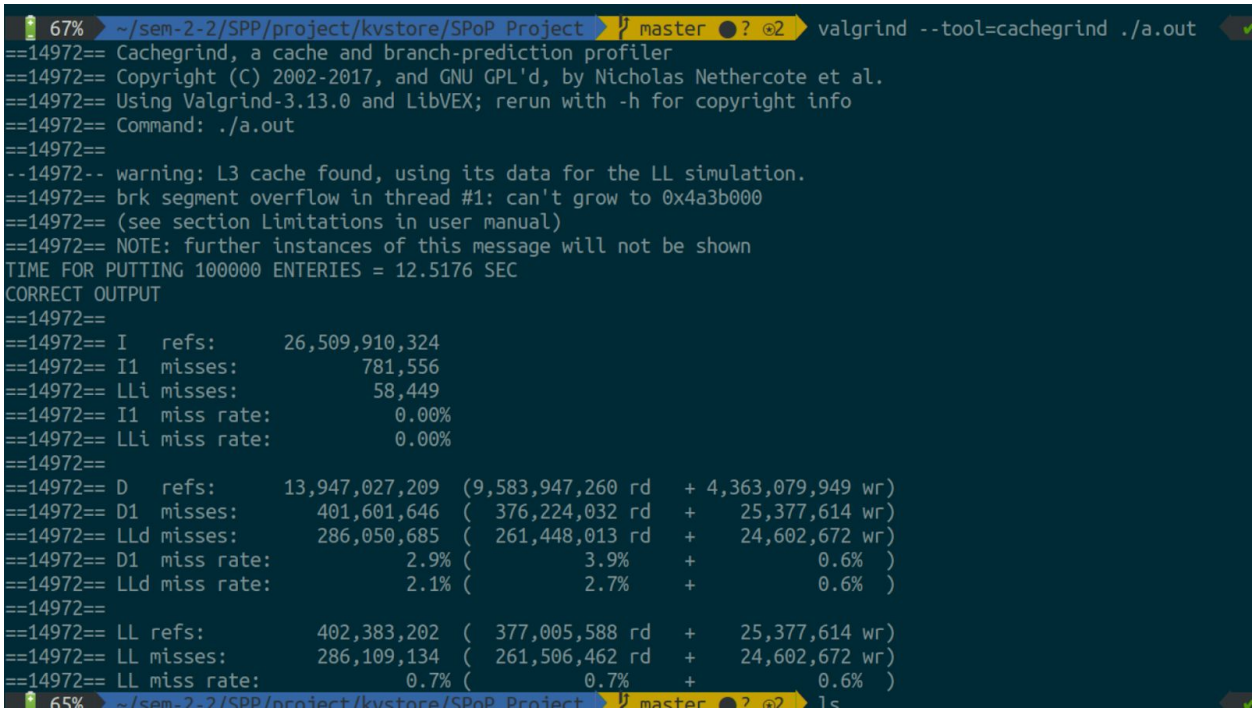
Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface. The perf tool offers a rich set of commands to collect and analyze performance and trace data.

```
 69% ~/sem-2-2/SPP/project/kvstore/SPoP Project  master ● ? ⊛2  sudo perf stat ./a.out
[sudo] password for sachin:
TIME FOR PUTTING 100000 ENTERIES = 1.0344 SEC
CORRECT OUTPUT

 Performance counter stats for './a.out':

     46584.872869      task-clock (msec)         #    0.999 CPUs utilized
              443      context-switches          #    0.010 K/sec
                0      cpu-migrations            #    0.000 K/sec
        3,81,781      page-faults               #    0.008 M/sec
  72,38,90,89,433      cycles                    #    1.554 GHz
  27,80,39,16,621      instructions              #    0.38  insn per cycle
   6,84,64,72,490      branches                  #  146.968 M/sec
     19,31,25,483      branch-misses             #    2.82% of all branches

     46.615653834 seconds time elapsed
```

Cache Grind :

Cachegrind is a tool for doing cache simulations and annotating your source line-by-line with the number of cache misses. In particular, it records:

- L1 instruction cache reads and misses;
- L1 data cache reads and read misses, writes and write misses;
- L2 unified cache reads and read misses, writes and writes misses.



Gprof :

Gprof calculates the amount of time spent in each routine.

```
71% ~/sem-2-2/SPP/project/kvstore/SPoP Project master ● ? ⊙2 cat profile-data.txt | grep 'kv.\| self'
 %   cumulative   self              self     total
11.21      1.72     0.26  3035920     0.00     0.00  kvStore::getNode()
 2.16      2.14     0.05   102001     0.00     0.00  kvStore::put(Slice&, Slice&)
 1.72      2.27     0.04     1951     0.02     0.02  kvStore::del(int)
 0.00      2.32     0.00     7872     0.00     0.00  kvStore::get(Slice&, Slice&)
 0.00      2.32     0.00     2128     0.00     0.00  kvStore::get(int, Slice&, Slice&)
 0.00      2.32     0.00     1986     0.00     0.00  kvStore::del(Slice&)
 0.00      2.32     0.00        1     0.00     0.00  kvStore::kvStore(unsigned long)
 self      the number of seconds accounted for by this
 self      the average number of milliseconds spent in this
index % time    self  children    called     name
                0.05    0.26  102001/102001      kvStore::put(Slice&, Slice&) [5]
                0.04    0.00    1951/1951        kvStore::del(int) [17]
                0.00    0.01    2128/2128        kvStore::get(int, Slice&, Slice&) [28]
                0.00    0.00    7872/7872        kvStore::get(Slice&, Slice&) [114]
                0.00    0.00    1986/1986        kvStore::del(Slice&) [126]
[5]     13.4    0.05    0.26  102001         kvStore::put(Slice&, Slice&) [5]
                0.26    0.00 3035919/3035920     kvStore::getNode() [7]
                0.00    0.00       1/3035920     kvStore::kvStore(unsigned long) [44]
                0.26    0.00 3035919/3035920     kvStore::put(Slice&, Slice&) [5]
[7]     11.2    0.26    0.00 3035920         kvStore::getNode() [7]
[17]     1.7    0.04    0.00    1951         kvStore::del(int) [17]
[28]     0.2    0.00    0.01    2128         kvStore::get(int, Slice&, Slice&) [28]
                0.00    0.01    2128/2128        kvStore::get(int, Slice&, Slice&) [28]
                0.00    0.00       1/1           kvStore::kvStore(unsigned long) [44]
[44]     0.0    0.00    0.00       1         kvStore::kvStore(unsigned long) [44]
                0.00    0.00       1/3035920     kvStore::getNode() [7]
[114]    0.0    0.00    0.00    7872         kvStore::get(Slice&, Slice&) [114]
[126]    0.0    0.00    0.00    1986         kvStore::del(Slice&) [126]
```

Call Graph :

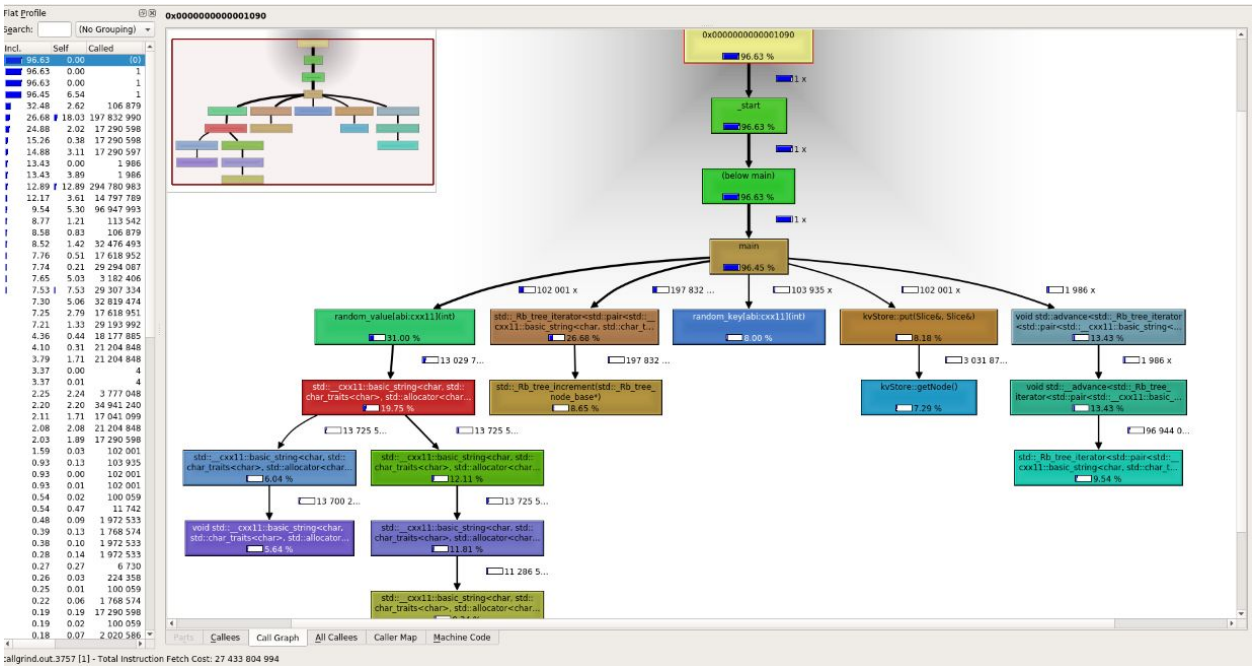
This graph explains the performance of the functions in terms of memory usage.

Command to generate call graph:

Valgrind --tool=callgrind ./a.out

kcachegrind callgrind.out.*

The call graph gives us the overview of the timing information of each function call.



TPS :

These are the throughputs we observed

```
vinay@vinay-Inspiron-5570:~/sem4/spp/project/kvstore/SPoP Project$ ./a.out
TIME FOR PUTTING 100000 ENTERIES = 0.832213 SEC
CORRECT OUTPUT
TRANSACTIONS PER SEC = 10000
TRANSACTIONS PER SEC = 10000
TRANSACTIONS PER SEC = 9800
TRANSACTIONS PER SEC = 10000
vinay@vinay-Inspiron-5570:~/sem4/spp/project/kvstore/SPoP Project$ ./a.out
TIME FOR PUTTING 100000 ENTERIES = 0.832005 SEC
CORRECT OUTPUT
TRANSACTIONS PER SEC = 9800
TRANSACTIONS PER SEC = 9800
TRANSACTIONS PER SEC = 9700
TRANSACTIONS PER SEC = 9700
vinay@vinay-Inspiron-5570:~/sem4/spp/project/kvstore/SPoP Project$ ./a.out
TIME FOR PUTTING 100000 ENTERIES = 0.832009 SEC
CORRECT OUTPUT
TRANSACTIONS PER SEC = 10100
TRANSACTIONS PER SEC = 10200
TRANSACTIONS PER SEC = 10100
TRANSACTIONS PER SEC = 10000
```