# Log Structured Merge Tree

## Part 1: Design of the project

### Brief introduction of this project

Log Structured Merge Tree(LSM) is a kind of data structure which can store huge amount of data based on the log of operation. The whole system can be divided to two parts, memory and disk. When a record is inserted, it will stay in memorey temporarily. Data structure in memory can be AVL Tree and Red-Black Tree. But in this project, jump table is used to realize the memory part. When the scale of jump table reaches a certain threshold, program will move data from memory to disk where data is organized in the form of SSTable. SSTable is a long string stored in th local disk. The first half of SSTable is data where key and value are stored. The second half of SSTable is index of data containg key and offset of key-value pair. Data in disk is divided into different levels whoes number of SSTables increses exponentially accroding to the index of level.

More ditail will be introduced in the following part.

### My understanding of the system

According to previous introduction, data inserted is firstly stored in memory. This is because the speed of accessing memory is much more faster than that of accessing disk. This method of storing data in memory first and saving it to disk together reduces the number of disk accesses effectively.

But with the number of inserted data increasing, the scale of SSTable will be larger too which is not a good news for query continuously. So SSTables are divided into different levels. Each level has its own file number limitation. The lager the index of level, the greater number of files.

All the SSTable generated for memory is stored firstly in level0. When file number exceeds limitation, exceeded files are compacted and move to the next level. By doing this recursively, the number of files in each level returns to normal. When user does a query, program will check data in memory first. If the key doesn't exist in memory, progrma will scan data in levels from top to bottom. Because of the time stamp added to the index part of SSTable and top-down consolidation, key finded first is always the newest key. So, we don't need to worry about repeat insertion.

Besides, deletion is similar to insertion. We just need to insert an empty stirng to cover the old entry.

### My unique implementations

- The memory part of program uses jump table. Binary number is used to encode SSTable. All data is stored as binary number in SSTable, just like that in RAM. Before writing integer to SSTable, I need to cast the pointer of integer to char pointer like `out.write((char *)(&key), 8)`. When I need to extract integer from SSTable, I will use the following code `indexOffset = *((int *)SSTInfo);`. All of this is trick of using pointer.
- Besides, I add some extra infomation to the end of SSTable so that I can get scale and offset of index quickily with the help of function `seekg(-8, ios::end)`.

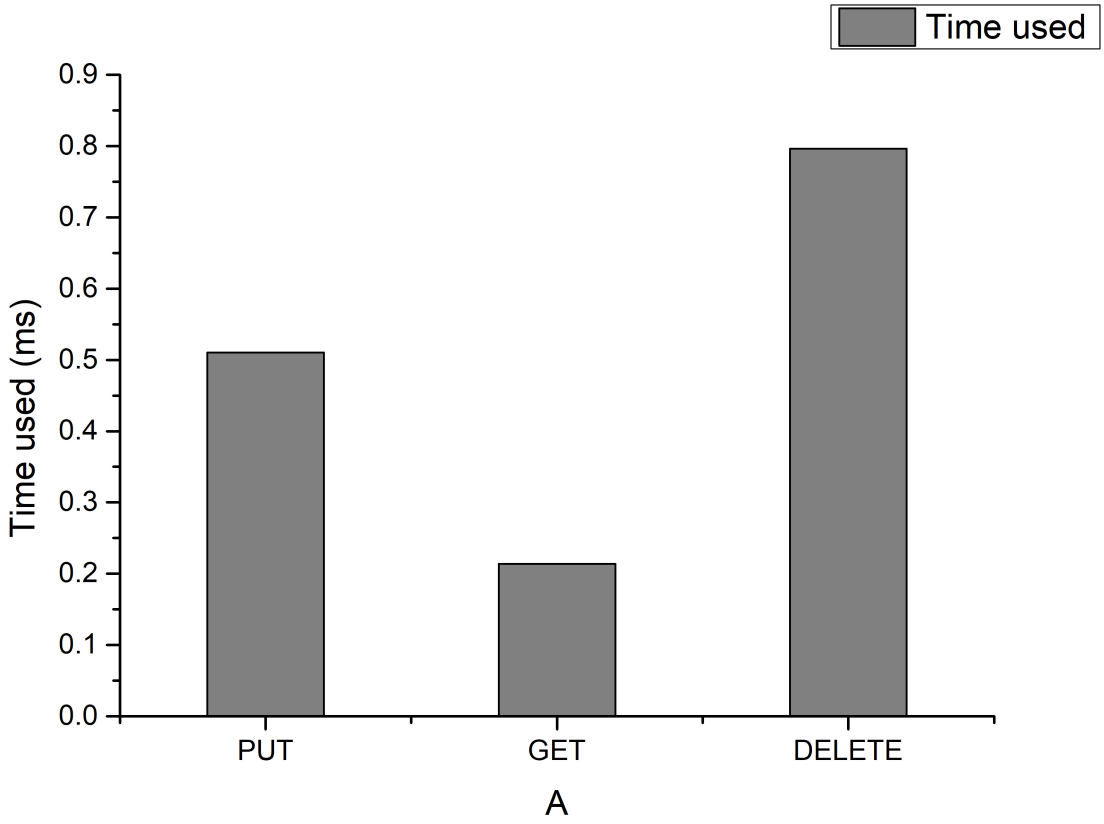
- When doing compaction, I use class `Buffer` to simulate file operation. The only difference between `Buffer::write(char *, int bytes)` and `Buffer::write(char *, int bytes)` is that the former happens in RAM and the latter happens on disk. As what is said priviously, RAM operation is much faster than disk operation.
- When doing a qurey on disk, program will check whether there is a copy of index in `vector<dataInfo *> Buffer::dataSet`. If so, program will search the copy using binary search. Else, program will access the file on disk and make a copy from disk to RAM.

# Part 2: Test Report

- **Test environment:** Windows

### Test 1: Average time of single operation without compaction

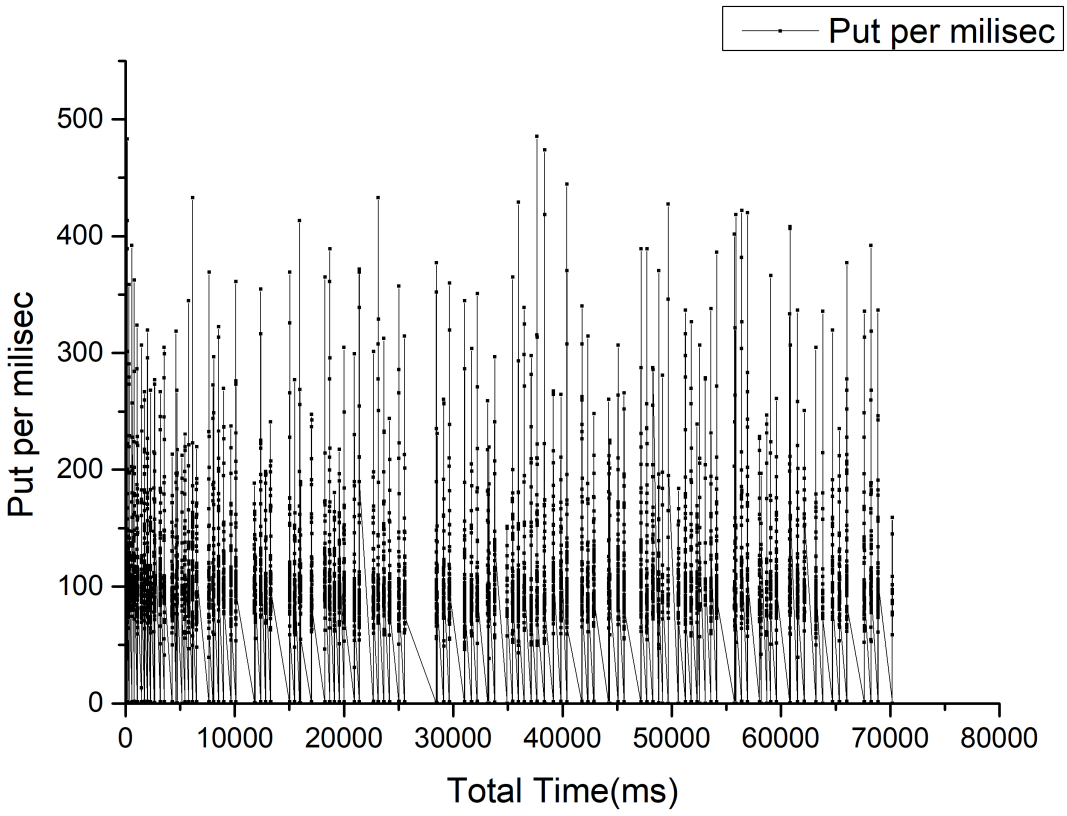
- **Test argument and method:** By putting/getting/delete 80000 strings which is "8888888" and record the interval. Then use the total time to divide 80000.
- **Test result:**




- **Analysis:** In this test case, we see that delete operation spends the most time. And get operation spends the least time. Put operation is somewhere in between. Because delete operarion needs to query the key first then do the deletion(insertion empty string). So, theoretically, time spent by delete operaion is the sum of that spent by put operation and get operation, which is proved by the graph.

### Test 2: Put per unit of time

- **Test argument and method:** Put 60000 keys totally by code `s.put(num,string(10000,'s'))`. Each 10 put an interval will be record. So, finally we can get 600 records(6000 / 10 = 600). We can use "10 / interval" to compute put operation per unit of time. **P.S.** Unit of time is millisecond in this test case.
- **Test result:**




- **Analysis:** In this grapg, we can see that put operation per millisecond fluctuates dramatically over time. By a closer view, I found that it can be divided into three parts horizontally. The top part represents operations happens in RAM which is the fastest. The middle part happens when jump table needs to be transfer to SSTable which invloves disk operation. So, the middel part is slower than latter. The bottom part is the slowest. This part happens when the number of SSTables exceeds the limitation of its level. And compaction means lots of disk operations. So, in this part operations per millisecond is closed to zero.