Implementation of LSM Trees for key value data stores

Mansi Madhvani	201901194
Abhiram Bhimavarapu	201901266

Summer Research Internship
May 2022 - July 2022
Guide : Prof. PM Jat

Objective/Motivation:

The main purpose of this project is to learn and deep dive into the working and internal structure of LSM Trees. We learn the functionalities of LSM Trees and implement it for the key-value data stores in C++. We also analyze the advantages and disadvantages of LSM Trees.

Theory:

Log-Structured Merge Tree (LSM Trees) is a very fundamental data structure in the field of computer science. LSM Trees are known for their exceptional high write throughput. They are mostly used while dealing with heavy-write loads. This important feature makes LSM Trees the core data structure in the internal implementation of most highly scalable NoSQL distributed key-value type databases.

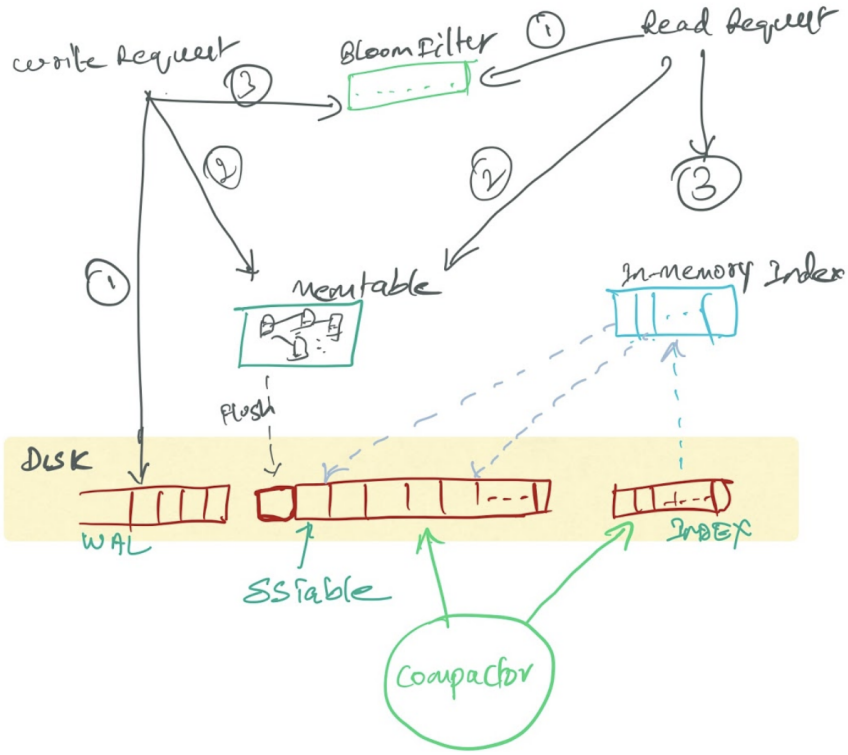
LSM Trees is a method of maintaining real-time data/index at a low cost. It is a disk-based data structure which is designed to provide low-cost indexing for files experiencing high inserts [PUT()]. The put request is performed only “in memory”.

In contrast to B+ Trees which perform the insert function directly to the disk, making it expensive, LSM Trees use memory based insertion making it a high write throughput data structure.

Internal Structure of LSM Trees:

The main components in the structure of LSM Trees are:

- MemTable
- SS Table
- Compactor
- Index
- Bloom FIlter



Source: https://medium.com/swlh/log-structured-merge-trees-9c8e2bea89e8

MemTable:

Every read or write operation performed is first directed towards the MemTable. It is an "**in-memory**" structure which buffers the incoming write requests before shifting it to the disk. After holding a certain amount of threshold data, Memtable will then shift the data. This threshold data will depend on the internal implementation of the Memtable and is mainly limited by RAM. All the write requests are sorted using an efficient self balancing binary tree and then shifted to the disk.

SS Table:

SS Tables mainly acts as the disk where the value is stored when the MemTable reaches its maximum capacity. They are the file formats used when MemTables are shifted to durable storage from memory. The data in the SS Table is stored in a sorted manner to facilitate efficient read performance.

Index:

The main purpose of maintaining an index is to make the read requests easy and efficient. These indexes contain keys and offsets in sorted order. Hence, these indexes are used to

quickly find the offset for values that would come before and after the key we want. This makes it easier to search in a small portion of the offset.

Compaction:

Compaction is an important part in the LSM Trees. The performance of read and write are greatly impacted by compaction. Compaction is basically a method to clean multiple occurrences of keys in the SS Tables. Compactor merges the SS Tables by removing any redundant and deleted keys and creating a merged SS Table.

Bloom Filters:

Bloom Filter is a probabilistic data structure which checks if a key is present or not in the system. It has a time complexity of O(1).
There are two possibilities:

- False Positive match - It may indicate presence of a key although it is not in the system.
- False Negative match: It may indicate a key is not present, then it is definitely not present. This helps in avoiding the expensive read path

Implementation:

To implement LSM Trees, we require two types of storage namely memory and disk.
A map data structure will be used to indicate a memory because of its fast retrieving nature. A 2D dynamic array (Vector) will be used to indicate the disk space.

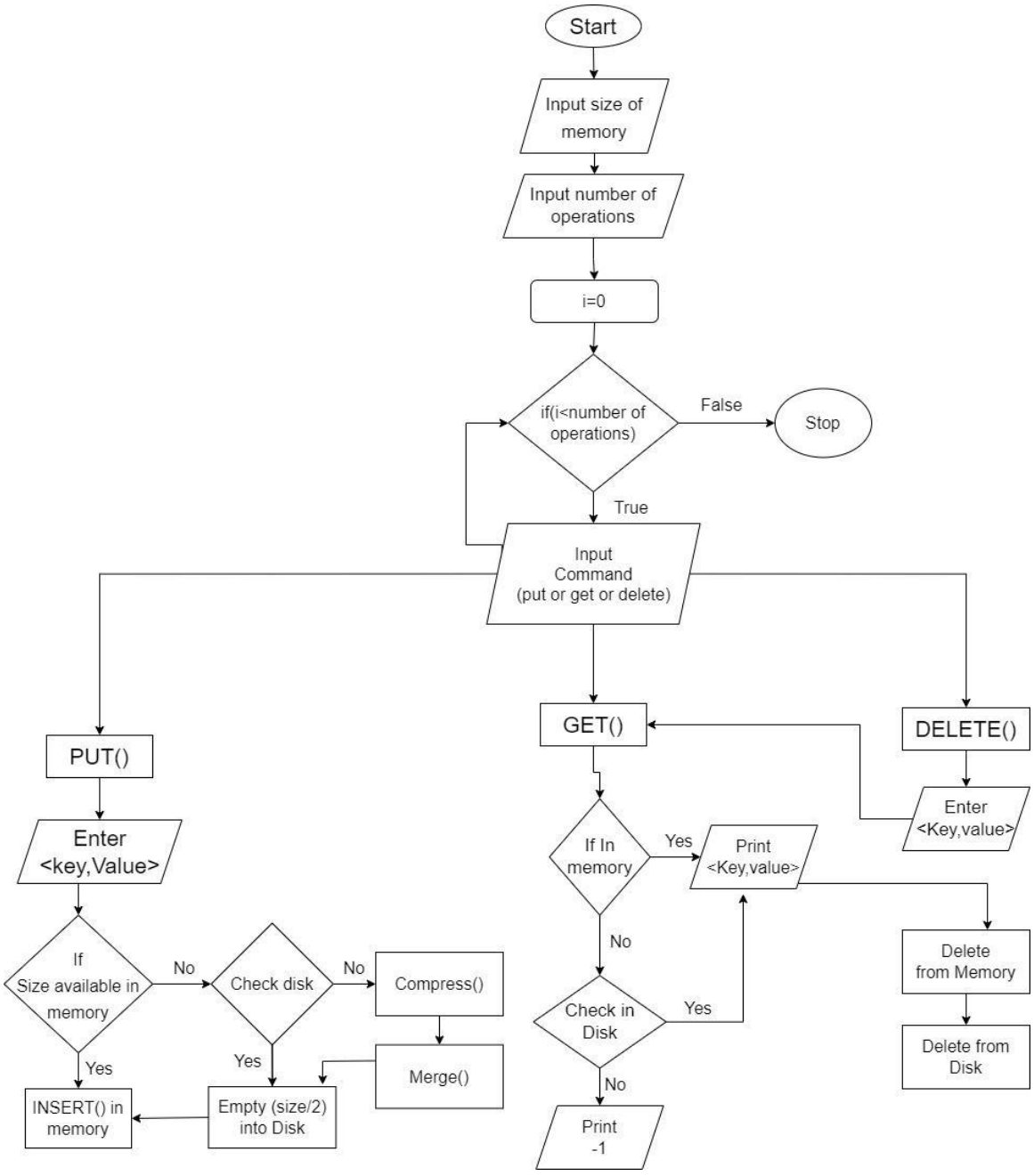
In the map (which acts as a memory), a <key, value> pair is stored until a certain threshold memory is reached. It is denoted by sizeOfMemory.

Next, we take a 2D vector (which acts as a disk), essentially a contiguous array containing another set of arrays denoting chunks and can store large amounts of data. Hence, disk is represented as vector<vector<keyValueData>> segments (2*sizeOfMemory+1). The ith node is denoted by segments[i]. The data stored at these nodes can be accessed by segments[i][j].

We have three main operations in our implementation

- Put() - inserts a key value pair
- Get() - searches a given key and gives the value corresponding to that key.
- Delete() - Deletes all the occurrences of key and its corresponding values.

Flowchart of the implementation:



Pseudo Code:

Put():

- Operation: PUT/put.
- Enter the key and values respectively.
- Check if there is a space available in the memory.
    - If space available, then insert the <key, values> in it.
    - If no space available, empty the first (SizeOfMemory)/2 pairs to the disk.
- Check if the disk is empty
    - If disk not empty, apply compress()
        - In compress function, Merge all the disk chunks into one node.
        - Call the merge function to merge the disk nodes to node 1.
- After compaction, free the first half of the memory in node 2.
- Insert the data into our disk node 2 erase first half of the data from memory.
- Exit the compress function and insert the <key, value> into the memory.
- Update the next free node to node 3.

Get():

- There are two search functions - searchInMemory, searchInDisk.
    - searchInMemory()
        - Take the input arguments as sizeOfMemory and key to be found.
        - As map is implemented by a red black tree internally it takes O(sizeOfMemory) to search the key in memory.
        - If not found, return “Not found in memory”.
        - If found, return the value at the key.
    - searchInDisk()
        - Take the input argument as segment[], key to be found and available node.
        - Here the search is performed from the latest used segments to the oldest ones.
        - If not found, return “Not found in disk”.
        - If found, return the value at the key.

Delete():

- There are two delete functions - deleteInMemory, deleteInDisk.
  - deleteInMemory()
    - Take the input arguments as sizeOfMemory and key to be deleted.
    - If count of the key in the map data structure (memory) is 0 then return “Not Found”
    - If count ≠ 0, erase all instances of the key inside the memory.
  - deleteInDisk()
    - Check all the nodes on the disk.
    - Let p1 = lower bound, p2 = upper bound.
    - p1 will give the oldest key instance and p2 will tell the newest key instance.
    - All the instances will be deleted from p1 to p2.

Output:

Put():

```
Enter the size of memory
5
Enter the total number of operations to be performed
15
```

```
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
Hello
Enter value
1
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
World
Enter value
2
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
This
Enter value
3
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
Is
Enter value
4
```

```
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
Implementation
Enter value
5
```

Get():

```
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
get
Enter key
Hello
The value found in the memory at Hello is 1
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
of
Enter value
6
```

```
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
LSM
Enter value
7
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
get
Enter key
Hello
The value found in the disk at Hello is 1
```

Delete():

```
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
put
Enter key
Hello
Enter value
6
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
get
Enter key
Hello
The value found in the memory at Hello is 6
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
del
Enter key
Hello
The value Hello is deleted successfully from memory.
The value Hello is deleted successfully from the disk.
Enter the type of operation to perform ( PUT- Insert the value, GET- Search the value, DEL-Delete the value )
get
Enter key
Hello
Value not found in the memory and in disk hence returning -1
```

Conclusion:

With the implementation of the LSM Trees, it is clear that it provides better write performance at a low cost. The append only happens in the MemTable makes LSM trees more reliable. This method is used over B+ trees as it provides high write throughput. Many big companies have shifted to LSM trees for the same reason. For example, Yahoo has steadily progressed from read-heavy workloads to read-write workloads. LSM implementations such as LevelDB and Cassandra have provided better write performance and companies are moving towards it gradually.

Further work has also been carried out in LSM Trees. Yahoo has developed a system called Pnuts which integrates both LSM trees and B+ Trees. IBM and Google have also done work in the similar path using different approaches.