Advanced Data Structures
(UE22CS642A)

Assignment Report (Skiplist & LSM Tree)

Team No: 22

Team Members:

J Thanish Vishaal (PES1PG23CS019)

Prajwal K B (PES1PG23CS053)

Class: MTech Sec 'A'

Date: 17/05/2024

❖ Datasets:

- First, we write a python script to generate 10 million key-values by using the python libraries like Random and String.
- Then we generated a key-value pair of 10 million with a key of length 5 and a value of length 20, and saved that in a file named as 'Random_10M_kv_pairs.txt'.
- Key - value pair is a combination of english alphabet and digits by using the random.choices(string.ascii_letter + string.digits).
- Create a subset of 1 million, 10k ,100k random key-value pairs from a larger dataset containing 10 million key-value pairs.
- The input data is sourced from the file Random_10M_kv_pairs.txt, which contains 10 million key-value pairs.
- The input data is read into memory, and the lines are shuffled randomly using the random.shuffle() function from the Python random module. This ensures that the selected subset is representative and not-biased by the original order of the pairs.
- After shuffling, the first 1 million lines are selected from the shuffled list. These lines represent the subset of 1 million random key-value pairs.
- The selected subset of key-value pairs is written to a new file named Random_1M_kv_pairs.
- And did the same for 10k ,100k sub-dataset.

❖ Hardware Configuration:

- **RAM:** 16GB
- **CPU:** AMD RYZEN 7
- **GPU:** Nvidia RTX 3060
- **ROM:** 2TB

❖ Software Configuration:

- **Python:** 3.x
- **Anaconda:** 24.3
- **Jupyter Notebook:** V7.0.8

❖ Skip List Implementation:

A. Overview of Project:

- The provided code is a Python implementation of the Skip List data structure.
- Skiplists serve as a probabilistic alternative to balanced trees, offering an average-case time complexity of O(log n) for search, insertion, and deletion operations.
- This efficient performance is achieved through a unique approach that allows the skiplist to skip over certain nodes during traversal, effectively reducing the number of comparisons required to locate or manipulate elements within the structure.

B. Features:

- **Insertion:** Inserting elements with random key values.
- **Deletion:** Deleting elements by key value.
- **Search:** Searching for a particular element by its key value.
- **Range queries:** It enables querying elements within a specified key range.

C. Requirements:

- Python3.x
- Jupyter Notebook

D. Performance:

- The Skip List implementation offers efficient search, insertion, and deletion operations, with an average-case time complexity of O(log n).
- However, its memory consumption remains a notable limitation, particularly when dealing with larger data sets. As the size of the data set increases, the Skip List's memory footprint can grow substantially, owing to the dynamic allocation of pointers and the potential for nodes to have multiple levels with unused forward pointers.
- Consequently, this characteristic of the Skip List implementation may pose challenges when working with resource-constrained environments or when handling exceptionally large data sets.

❖ LSM Tree Implementation:

A. Overview of Project:

- This project is an implementation of a Log-Structured Merge-Tree (LSM Tree) data structure in Python. LSM Tree is a disk-based data structure optimized for providing fast insertions, deletions, and range queries on large datasets.

B. Features:

- **Insertion:** It allows for efficient insertion of key-value pairs into the data structure.
- **Deletion:** It supports the deletion of individual keys or a range of keys from the LSM Tree.
- **Range Queries:** It enables querying a range of keys efficiently.
- **Search:** It searches for a particular element by its key value.
- **Memory Management:** It ensures efficient memory usage by employing memtable (in-memory components) and sorted_nums (on-disk components).

C. Requirements:

- Python3.x
- Jupyter Notebook

D. Performance:

- The LSM (Log-Structured Merge) Tree implementation offers efficient insertion and deletion operations. The time complexity for operations on the memtable (in memory component) is O(log n) on average, while for the SSTables (on-disk components), it is O(k), where k represents the number of SSTables that overlap with the query range.
- However, the memtable's memory consumption can become a bottleneck, particularly for larger datasets. The performance of range queries is influenced by the number of SSTables that intersect with the query range, as more SSTables need to be scanned to retrieve the desired data.

❖ Time Complexity (In seconds):

	Skiplist				Lsm Tree			
	Insert	**Delete**	**Search**	**Range Query**	**Insert**	**Delete**	**Search**	**Range Query**
10K	0.165	0.022	0.003	0.001	0.035	0.001	0.00097	0.00099
100K	1.384	0.147	0.03	0.00099	0.4901	0.001	0.0089	0.0490
1M	17.692	1.456	0.2342	0.004	13.01	0.222	1.234	0.448
10M	280.988	13.009	2.5323	0.0645	169.65	2.33	2.56	0.565

❖ Space Complexity (In KB):

	Skiplist				Lsm tree			
	Insert	**Delete**	**Search**	**Range Query**	**Insert**	**Delete**	**Search**	**Range Query**
10K	283.004	282.976	281.758	34.749	283.146	283.118	-	-
100K	2832.002	2831.974	1740.594	1222.304	2831.9746	2831.97459	-	-
1M	28320.284	28320.255	12655.781	3923.779	104243.2	104243.09	-	-
10M	283203.125	283203.096	147617.504	46842.589	210688.5	210687.9	-	-