Optimized Database Design for Write-Heavy Workloads

Chinmay Bhusari - 21CS30015
Ronit Nanwani - 21CS30043
Apoorv Kumar - 21CS10008
Sanchit Yewale - 21CS10056

Optimized Workload

Our database is optimized for write-heavy workloads, making it well-suited for Online Transaction Processing (OLTP) systems. In this setup, key-value pairs are stored in a text file, and an index is created for efficient data retrieval. The database is particularly effective when frequent insertions and updates are required, as it supports high transaction volumes. With the use of techniques like Log-Structured Merge (LSM) Trees, the system ensures that writes are processed efficiently while maintaining fast lookup capabilities. This design is ideal for applications that require high write throughput and consistency, with mechanisms in place to ensure data durability and reliability.

Data Structures Used

AVLTreeNode Struct

Purpose: Represents a node in the AVL Tree.

AVLTree Class

Purpose: Implements the AVL Tree itself and provides methods to insert, erase, and search for elements. It also manages the balancing of the tree using rotations.

The insert and erase methods use a vector path to track the path from the root to the point where insertion or deletion happens. After performing an operation, the tree is balanced starting from the point of insertion/deletion upwards along this path.

In-order Traversal:

The getSortedPairs() method performs an in-order traversal of the tree. In-order traversal visits nodes in ascending order of their keys, making it useful for obtaining the sorted sequence of key-value pairs.

ProbabilisticSet

The ProbabilisticSet class implements a probabilistic data structure similar to a Bloom Filter. It efficiently determines if an element is a member of a set with a certain probability of false positives and no false negatives.

Data Members:

- `bitVector (bitset<ARRAY_SIZE>)`: A bit vector of size 10 used to represent the set. Each bit indicates whether a corresponding position is set (1) or not (0).
- `numHashFunctions (int)`: The number of hash functions used to generate multiple hash values for each key. It is calculated based on the size of the bit vector and the expected maximum number of items in the set.
- `multipliers (vector<size_t>)`: A list of constants used to create unique hash values for the same input key, ensuring that each hash function generates distinct indices for the bit vector.

Functions

- `encodeKeyValuePair()`: Concatenates a key and value with a delimiter, creating a combined string representation of the pair.
- `decodeKeyValuePair()`: Splits the combined string back into its key and value based on the delimiter.
- `splitString()`: Splits a string into tokens based on a delimiter, returning them as a vector of strings.
- `extractPair()`: Extracts a pair of integers from a specific position in a binary file, reading the key and value pairs stored there.
- `extractKeyValuePair()`: Reads key-value pairs from a binary file, stopping when two delimiters are encountered.
- `createFolder()` and `deleteFolder()`: These functions handle folder creation and deletion, useful for managing SSTable directories.

SSTable Class

The SSTable class stores key-value pairs in an ordered and compact format. It uses a Bloom filter for fast key existence checks, binary files for efficient storage, and provides methods for finding, storing, and retrieving key-value pairs.

Constructor: Takes a pair of data (key-value pairs) and a folder name. If the folder name is not provided, it generates a new folder name based on the current SSTable list size.

Destructor: Cleans up by deleting the associated folder.

Methods:

- `find()`: Searches for a given key in the SSTable using a binary search approach.
- `store_keyval_index()`: Stores index data (key positions) in binary files.

- `store_keyval_data()`: Stores the actual key-value pairs in text files, splitting them across multiple files if necessary.

Compaction and Synchronization

`create_SSTable()`: Creates an SSTable from a list of key-value pairs, performs compaction, and adds the SSTable to the list.

Synchronization: Semaphores (`sem_compaction` is used to synchronize access to shared resources like the AVL tree and the SSTable list. This ensures thread safety when performing compaction or inserting/deleting keys.

SET, DEL, and GET Operations

- `SET()`: Inserts a key-value pair into the AVL tree. If the tree exceeds a defined size (`MAX_TREE_SIZE`), the data is moved to a new SSTable.
- `DEL()`: Marks a key as deleted by inserting a tombstone (a special "deleted" value).
- `GET()`: Retrieves the value associated with a given key. It first checks the AVL tree, and if the key isn't found, it checks the SSTables in reverse order (newest first).

Concurrency Control

We have implemented a concurrency control mechanism between the compaction thread and the main database thread by utilizing a simple mutex. This mutex ensures that access to shared resources is synchronized and prevents race conditions. The primary shared resource between these threads is the SSTable list, which holds the data structures related to the sorted string tables (SSTables). By using the mutex, we ensure that only one thread can modify or read from the SSTable list at any given time, thereby maintaining data integrity and preventing conflicts.

Workflow

1. **Key-Value Pair Encoding and Decoding:**
   - The `encodeKeyValuePair` function combines the key and value into a string separated by a delimiter.
   - The `decodeKeyValuePair` function splits the combined string back into the key and value based on the delimiter.
2. **String Splitting:**
   - The `splitString` function breaks a string into tokens based on the delimiter for further processing.
3. **File Reading Operations:**
   - The `extractPair` function extracts an integer pair (key, value) from a binary file at a specific index.

- The `extractKeyValuePair` function retrieves a key-value pair from a binary file at a specified position.

4. **Folder Operations:**
   - `createFolder`: Creates a folder if it does not already exist.
   - `deleteFolder`: Deletes a folder and its contents.

5. **SSTable Class:**
   - Constructor: Initializes an SSTable, creating the folder and storing key-value pairs along with their corresponding Bloom filter and indices.
   - Destructor: Deletes the SSTable folder.
   - `find`: Searches for a key in the Bloom filter and binary search within the SSTable files to find the corresponding value.
   - `store_keyval_index`: Stores key-value index pairs in binary files.
   - `store_keyval_data`: Stores key-value pairs in text files, ensuring that file size constraints are respected.

6. **SSTable Creation:**
   - `create_SSTable`: Converts a vector of key-value pairs to a dynamically allocated array and stores it in a new SSTable. It triggers compaction if necessary.

7. **SET Operation:**
   - The `SET` function inserts a key-value pair into the AVL tree. If the tree size exceeds the threshold, it moves data into an SSTable.

8. **DEL Operation:**
   - Marks keys as deleted by inserting a tombstone into the AVL tree.

9. **GET Operation:**
   - Retrieves the value associated with a given key from the AVL tree or SSTable.