Team members:

- **Lieyang Chen** (lc3548@columbia.edu)
- **Mengwen Li** (ml4643@columbia.edu)
- **Shengtan Mao** (sm4954@columbia.edu)

What would you like to produce

A key-value storage engine library utilitzing `SSTable` and `Log-Structured Merge-Tree (LSM-Tree)`.

We would like to using modern C++ to implement a key-value storage engine library providing `get(<key>)`, `set(<key>, <value>)` and `delete(<key>)` apis.

The library can be used as the base of a distributed NoSQL database that require persistent datastore.

The library is planned to include the following components:

- `MemTable`: An in-memory map that temporarily holds the key-value pair that user want to set through `set` api, and also the first place `get` will check to retrieve the value.
- `SSTable (Sorted String Table)`: Represents the file that stores the data on disk. Once `MemTable` reaches a certain size, it is flushed to disk in the format defined by `SSTable`.
- `SSTableIndex`: In-memory sparse index for a `SSTable`. This will be implemented to speed up `get` time since with index, we don't have to load the whole `SSTable` into memory to locate a record.

  `The \`SSTableIndex\` uses \`<key>\` as key, and the location on disk corresponding to this key as value.`
- `TransactionLog`: An append-only log for all the `set` and `delete` operations that are currently in `MemTable`. `TransactionLog` is used to make sure that nothing is lost if database crashes while there are still `MemTable` that's not flushed to disk. There will be one `TransactionLog` on disk corresponds to one `MemTable`, and once `MemTable` is flushed to disk, the corresponding `TransactionLog` will be deleted.

Illustration of the workflow:

- `set(<key>, <value>)`
  1. The corresponding `TransactionLog` will be updated to reflect this operation.
  2. `<key>` and `<value>` will be inserted into `MemTable`.
  3. If `MemTable` reaches a pre-defined size:
     a. Start an async job will be started to flush the `MemTable` to `SSTable` on disk. When flush is done:
        1. An `SSTableIndex` corresponding to the `SSTable` will be created.
        2. The corresponding `MemTable` is removed from the queue of `MemTable`s.
        3. The `TransactionLog` corresponding to this `MemTable` is removed from disk.
     b. A new `MemTable` will be insert into the queue to hold the incoming `set`.
- `get(<key>)`
  1. Check `MemTable`s in the queue to see if `<key>` exists. Return if it does.
  2. Check `SSTableIndex` from latest to oldest, if `<key>` falls in the range of two entries from an `SSTableIndex`:
     a. Load the block between these two entries from the corresponding `SSTable`.
     b. Go through all the entries from the block to see if `<key>` exists.
- `delete(<key>)`
  1. Insert a `tombstone` entry into `MemTable` to represent that this key is marked for removal.
- Async job to preform merging of `SSTable`s on disk to consolidate duplicate entries and removed entries
  1. An async job will be started to merge the `SSTable`s on disk when the number of tables reaches a certain size. When merging of the tables are done:
     a. Build index for the newly created `SSTable`.
     b. Remove all the `SSTableIndex`s that corresponds to the removed `SSTable`s.

How would you like to start

There are several things that could be done in parallel in the beginning:

1. Implement the `MemTable` and `set(<key>, <value>)` operation to write to the latest `MemTable`.
2. Implement the `SSTable` interface, define how the entries should be serialized to and deserialize from the file on disk.
3. Research on the metrics that we can use to profile our implementation, and setup leveldb to prepare for performance comparison.

   And also investigate additional features and optimizations that are done by other implementations the may contribute to performance difference.

Who will initially do what

- **Lieyang Chen (lc3548)**: Start implementing a program that can be used to test the library. Also researching other implementations of similar libraries to see for example what optimizations are done by those libraries.
- **Mengwen Li (ml4643)**: Start implementing serialization and deserialization of `SSTable`.
- **Shengtan Mao (sm4954)**: Start implementing `Memtable`, and also `TransactionLog`.

What will you eventually want to measure (quantify)

We plan to profile this library implementation by `operations per second` for the following operations:

1. A series of `set(<key>, <value>)` operations.
   a. With `<value>` with a small size.
   b. With `<value>` with a large size.
   c. With a mixture of small and large size `<value>`s.
2. A series of `get(<key>)` operations with `<key>` sorted. (serial access)
3. A series of `get(<key>)` operations with `<key>` in random order. (random access)
4. A mixture of `set(<key>, <value>)`, `get(<key>)` and `delete(<key>)` operations.

Other parameters that we could include in our profiling are:

1. Memory used while the program runs for a given amount of data stored.
2. Disk space used for a given amount of data stored.

Why do you think you can do it on this tight schedule

Conceptually, this project is relatively small in scope. It is a very basic database system. We believe the main difficulty of this project will be implementing the asynchronous operations of the database. This part is tricky to get right, but we think this is achievable with three people and also the new language support.

Reference

1. NoSQL Database Systems: A Survey and Decision Guidance
2. Bigtable: A Distributed Storage System for Structured Data
3. Chapter 3 of Designing Data-Intensive Applications
4. google/leveldb