Design Document

Introduction

`ProjectDb` is a key-value storage engine library, implementing the `Log-Structured Merge-Tree (LSM-Tree)` algorithm.

The library can be used as a local NoSQL database for programs that requires persistant storage of data in `[key, value]` format.

It can also be used as the base of a distributed NoSQL database. Although this library does not has any support for communicating between multiple nodes across the network that's running `ProjectDb`, such functionalities can be added latter as layers on top.

A brief description of `LSM-Tree` algorithm can be found in this wiki page.

Terminologies

- `MemTable`: This is a map resides in memory sorted by `key`. (Corresponds to `C0` in wiki).
- `TransactionLog`: For every `set(key, value)` and `remove(key)` operation we get, it will first be appended to a `TransactionLog` before inserting into `MemTable`. This is to make sure that when database crashes, we can recovery the entries that are in-memory (entries in `MemTable`) before crash happens.
- `SSTable`: This is the file created on disk once a `MemTable` reaches it's size limit. (Corresponds to `C1` in wiki).
- `SSTableIndex`: Each `SSTable` has a corresponding `SSTableIndex`. To build this index, we keep a `[key, <corresponding file location>]` in memory for every block size (default to `0.25 mb`, but can be configured by user).

  With this index, when user wants to get value for a give `key`, due to the fact that all `key`s are sorted, we don't need to load a whole `SSTable`. Instead, we can just load a specific block that we know this `key` might present.

  `ProjectDb` maintains a `m_ssTableIndexQueue` in memory to keep track of the `SSTableIndex` for all `SSTable`s.
- `TOMBSTONE`: When getting a `remove(key)` call, instead of actually removing the corresponding entry from the database, a `[key, TOMBSTONE]` entry is added (As a result, `remove(key)` will actually increase the disk space usage).

  This is because, with `LSM-Tree`, all operations are append-only, to achieve better performance.

  For `set(key, val)`, instead of going through the entire database to find `key` and reset its `value`, we will just append a new entry to `MemTable` and return. As a result, it is possible that we have multiple `key` with different `value` scattered in multiple `SSTable`s. So, for `remove(key)`, we will also not going to go through the whole database and remove all entries with `key`, we will instead just append a `TOMBSTONE` entry for this `key`. This ensures that all `set` and `remove` operations have `O(1)` time complexity.

  `compaction` will be done regularly to consolidate `SSTable`s, which will consolidate entries with the same `key`, as well as remove those entries that are marked as `TOMBSTONE`. The disk space usage will be reduced after `compaction`.
- `compaction`: The operation of merging multiple `SSTable` into one, to reduce both the total size of the files, and the number of files. The resulting `SSTable` after compaction corresponds to `C2` defined in wiki.

Design Decisions

- **Format of `key` and `value`**:

  With our current implementation, the format of `key` and `value` are both `std::string`. `std::string` is a generic enough representation for our use cases. And it's much easier to interface with comparing to `char*`.

  If user wants other types as `key` or `value`, they could do the encoding/decoding to/from `std::string` before calling `ProjectDb` apis.
- APIs that `ProjectDb` supports:

  The current version of `ProjectDb` provides 3 apis: `set(key, value)`, `get(key)` and `remove(key)`.

  We think that these are the apis, along with the combination of these apis, can satisfy most of the use cases of for a database.

  With further developments, more apis could potentially be provided, e.g. a `get` with a range provided by user.

- **Format of data on disk**:

  Everything is encoded into a list of `char` before flushing to disk. A more detailed description of the serialization format can be found in Documentation

  Because we don't expect a file written by `ProjectDb` on one machine to be opened on another machine, we don't have to worry about things like different machines having different size for `int`, endian differences between machines, etc.

  Also, the current serialization/deserialization implementation does not have a `VERSION_NUMBER` implemented for backward compatiblity. This is because it's still under heavily development. A `VERSION_NUMBER` will eventually be added to make sure that updates to `ProjectDb` will not cause old database to be not loadable. It's also relatively easy to add this `VERSION_NUMBER` with our current serialization/deserialization implementation.

- **Multi-thread / Multi-process support**:

  The library currently don't have multi-thread / multi-process support.

  For multi-thread usage, user could lock the `ProjectDb` object before calling apis with it. In later developments, we could add multi-thread support internally in `ProjectDb`, so that we could have a more granular control on what we need to lock.

  For multi-process usage, it is also currently not supported if they all has the same `DB_FILE_PATH` configured. This is because the database files could be written by multiple `ProjectDb` instances, and thus corrupted. In later developments, we could enable other process to perform `get(key)` operation with the same `DB_FILE_PATH`.

- **`SSTable compaction` algorithm**:

  The `SSTable` compaction algorithm that we decide to implement is described as following:

  It is controlled by config `NUM_SSTABLE_TO_COMPACT` and `SSTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES`.

  Internally we keep an `int m_compactionStartIndex`, which indicates which `SSTable` we should start compaction from. This variable will be updated everytime a compaction job finishes.

  A diagram explaining the variables (with `NUM_SSTABLE_TO_COMPACT == 10`) is as following:

```
  SSTable that already reaches SSTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES
  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ...
      |   |------ NUM_SSTABLE_TO_COMPACT ------|
      |
      m_compactionStartIndex
```

  Everytime we push a new `SSTableIndex` into `m_ssTableIndexQueue`, which means that there's a new `SSTable` being written to disk, we will check if we already have more than `NUM_SSTABLE_TO_COMPACT SSTable`s between current `m_compactionStartIndex` and the end of the queue. If so, we launch the `compaction` job for `NUM_SSTABLE_TO_COMPACT` number of tables.

  During compaction, we will always only have two `SSTable`s loaded in memory at a given time, denoted as `[oldTable, newTable]`. We will keep loading tables into `newTable`, and merge `newTable` with `oldTable` until `oldTable` reaches the size of `SSTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES`.

  At this point, we load the next table into `oldTable`, and continue the same process.

  The name of the merged table will be the same as the table that `oldTable` is loaded from. After this process is done, we flush all the merged table to disk with suffix `.merged` added, and get a list of corresponding `SSTableIndex`s. This is because we don't want to override the current `SSTable`s on disk, since users might be accessing them.

  During post-process of the `compaction` job (which is the first operation we do for all the apis), we will rename the deprecated `SSTable`s with `.deprecated` ext added, remove the `.merged` ext for the compressed `SSTable`s, and update `m_ssTableIndexQueue` with the new indices.

- **When to run flush `MemTable` to `SSTable`, and `compaction`, and how they should be done**:

  There are two operations that are expensive due to writing to filesystem:

  1. Flush `MemTable` (in memory) to `SSTable` (on disk) (`flush` for short).
  2. Merge `SSTable`s to reduce the number of files, as well as the total size of the files (Due to removal of duplicate and `TOMBSTONE` entries) (`compaction` for short).

Since these two operations could be very expensive, we can't let users wait for these when they call the apis that we provided.

As a result, we decided to launch these jobs using `async`, and store the returned `future`s. And when the apis are called, we go through these `future`s, for those that are finished, we do some post-processing which are very cheap.

For the `compaction` job, we can only run one at a time. This is because with our current compaction algorithm, the new compaction might depend on the result of the previous compaction.

For the `flush` job, it's possible that we launch multiple of them together. But, when processing the `future`s, we have to make sure to stop at the first non-finished one, which means that even if there are other `future`s after that are done, we should not process them. This is because we have to make sure that all the generated `SSTableIndex` are in order.

We decided to go with an approach that does not require any locking. As a result, it is important that we only do post-processing of `flush` job while there's no `compaction` job running. This is because post-processing of `flush` job updates `m_ssTableIndexQueue` to insert a new `SSTableIndex` entry, and `compaction` needs to read `m_ssTableIndexQueue` to get the file names for tables that needs compaction. If these run together in separate threads, there will be a race condition. However, this could be avoided if `compaction` job just takes in a list of file names instead of directly accessing `m_ssTableIndexQueue`. We will look into implementing this in future updates.

- **How many MemTable do we keep in memeory**:

  We decided to keep a queue of `MemTable`s in memory. Since `flush` is launched as an `async` job, it is possible that user continues to access the `MemTable` while we flush it to disk. However, since this `MemTable` already reaches its max size (`MEMTABLE_APPROXIMATE_MAX_SIZE_IN_BYTES`), we can't write to it. As a result, we have to insert another `MemTable` into the `MemTableQueue`.

  The size of the queue is related to how fast `MemTable`s are flushed to disk.