Document

This documentation provides a detailed description of directories, files, classes and methods for implementing `ProjectDb`.

Directory Structure

- `benchmark`: This directory contains the code used for benchmarking `ProjectDb`.
- `config`: This directory contains the code for parsing config files, as well as the header and cpp file for parameters used by the database.
- `db`: The `ProjectDb` related implementations, things like `Key`, `Value`, `MemTable`, `SSTable`, `TransactionLog`, etc are in this directory.
- `include`: This is the directory that needs to be added to the include path of `ProjectDb` user in order to access the apis provided.
- `tests`: This directory contains unit tests, written using `googletest` framework.
- `utils`: This directory contains utility class and functions.

Description of some files, classes and functions

- `.github/workflows`:

  This is used for setting up CI to run unit test and `clang-format` check everytime we create a pull request.

- `benchmark/*`:

  See Measurements.pdf for detailed description of benchmarks.

- `config`:
  - `db_config.h/cpp`:

    This file contains the parameters user can configure before running `ProjectDb`. A detailed description of each parameter can be found in comments of `db_config.h`.

    The performance of the database will be affected by some of the parameters.

  - `db_config_parser.h/cpp`:

    This file contains `ConfigParser` class that parses a config file that user provides into the parameters used by the database.

- `db`:
  - `key.h/cpp`:

    This file contains a wrapper class around `std::string` to represent the `key` type for the database.

  - `memtable.h/cpp`:

    This file contains `MemTable` class that represents a `MemTable`. Its methods represents the operations that could be done to a `MemTable`.

  - `memtable_queue.h/cpp`:

    This file contains `MemTableQueue` class that represents a queue of [`MemTable`, `TransactionLog`] pairs.

    When searching for a `key`, this is the first place we try.

  - `projectdb.cpp`:

    This file contains the implementation of the `ProjectDb` class.

    `ProjectDb` delegates all its implementation to `ProjectDbImpl` in order to minimize the implementation details exposed to user.

    It is put in `db` directory instead of together with `projectdb.h` under `include/projectdb` also to hide implementation details.

- `sstable.h/cpp`:

  This file contains `SSTableMetaData` and `SSTable` class to represent an `SSTable`.

  `SSTableMetaData` is something that we write to disk at the start of every `SSTable`. It currently only contains a timestamp, and is not being used. However, we decide to keep it here for further extensions.

- `sstable_index.h/cpp`:

  This file contains `SSTableIndex` class, that represents the index we generate for a given `SSTable`.

  `SSTableIndex` are generated at the same time we load or flush `SSTable` to disk.

- `sstable_index_queue.h/cpp`:

  This file contains `SSTableIndexQueue` class, that represents a queue of `SSTableIndex`. It contains indicies that maps to all the `SSTable`s on disk.

  When we failed to find a `key` in `MemTableQueue`, we will try to search for it in `SSTableIndexQueue`.

- `sstable_ops.h/cpp`:

  This file contains operations that we perform to flush, load, and compact (merge) `SSTable`s.

- `table.h/cpp`:

  This file contains a wrapper class around `std::map<Key, Value>` to represent the underlying table that we use to store the data. It is a member for both `MemTable` and `SSTable`.

- `transaction_log.h/cpp`:

  This file contains `TransactionLogWritter` class, which writes a `set` or `remove` operation to transaction log, and `TransactionLogLoader` class, which loads a transaction log into `MemTable`.

  `TransactionLogLoader` is needed during database initialization.

- `value.h/cpp`:

  This file contains a wrapper class around `std::string` to represent the `value` type for the database.

- `include/projectdb`

  - `projectdb.h`:

    This file contains `ProjectDb` class, which user will use to access the provided apis.

    This is the only file that user needs to include when using `ProjectDb`. For more details, see Tutorial.

- `utils`

  - `db_concepts.h/cpp`:

    This file contains the `concept`s and type traits we defined. These are mainly used to try to make serialization and deserialization more generic.

    A more detailed description can be found in comments in this file.

  - `exception.h/cpp`:

    This file contains `DbException` class, which is being thrown whenever there's an exception happens during operations.

    We decided to define our own exception class so that we can more clearly distinguish between the exception that's thrown by our code, and the those that are thrown by standard library for example.

  - `log.h/cpp`:

    This file provides utilities under `log` namespace that are used for printing logs.

    One main reason for adding logging utilities is that with `log::debug` wrapper for example, we can easily remove all debug log when we do a `release` build.

  - `serializer.h/cpp`:

    This file provides `SerializationWrapper` and `DeserializationWrapper` classes. These classes provide a generic, easy-to-use interface for doing serialization and deserialization.

Specifically, the `Serializable concept` is implemented following this defination:

```
SerializableBase = Trivial |
                   SerializableUserDefinedType

Serializable = SerializableBase |
               Pair<Serializable, Serializable> |
               Container<Serializable>
```

`Trivial` represents the POD types, and `SerializableUserDefinedType` represents user defined classes that have `serializeImpl` and `deserializeImpl` defined. (We did encounter this issue that a POD type could also have these two methods defined, resulting in an ambiguous match. Currently this is solved by adding a virtual dtor)

`Pair` represents a `std::pair`, and `Container` represents a container like class. (With the current `Container` concept defination, it also matches `std::string`)

With this recursive defination of `Serializable`, it can match every data structures and classs that we need to serialize and deserialize, things like `int` for timestamp, and `std::map<Key, Value>` for `Table`. And, due to the recursive defination, it can also match nested data structures, such as `std::map<Key, std::vector<Value>>`, which could be useful latter when more features are added to the database.

Regarding the format of data that we store on disk, since the data on disk is intended to be write and read on the same machine, we don't have to worry about things like POD size difference, or endian difference between machines.

Also, since the serialization and deserialization will be called by the same program (in normal cases), we don't have to worry about encoding the type information to disk, because deserializer should know what type should the bytes be deserialize to.

However, this will have a problem if user first build up a database using the current version, then, a newer version of `ProjectDb` is used to read this previously populated database. In this case, if there are some serialization/deserialization related updates, the deserialization might fail.

This could be resolved by adding a `VERSION_NUMBER` so that we get backward compatability. We will implement this in the future when the project is more stable.

Below lists the format we use for different type of data:

* `Trivial`: Conver to a vector of bytes.
* `SerializableUserDefinedType`: Just calls `serializeImpl` and `deserializeImpl`. So it's up to the user to define how it should be serialized/deserialized.
* `Pair`: We serialize `Pair.first`, then serialize `Pair.second`.
* `Container`: We first serialize `Container.size()`, so that we know how many entries to read. Then, we serialize each entry of the container.

- `tests`:

  This directory contains unit tests that are implemented using google-test framework.

- `Dockerfile`:

  This file is used to run CI on all the PRs that we submitted, as well as uniform the development environment across the team.

- `.clang-format`, `apply-format`, `git-pre-commit-format`, `init.sh`:

  These are files that are used for install git pre-commit hook to automatically run `clang-format` when commit.

  `apply-format` and `git-pre-commit-format` are taken from https://github.com/barisione/clang-format-hooks/