Ledis - An Implementation Report

Tech stack

1. Server-side

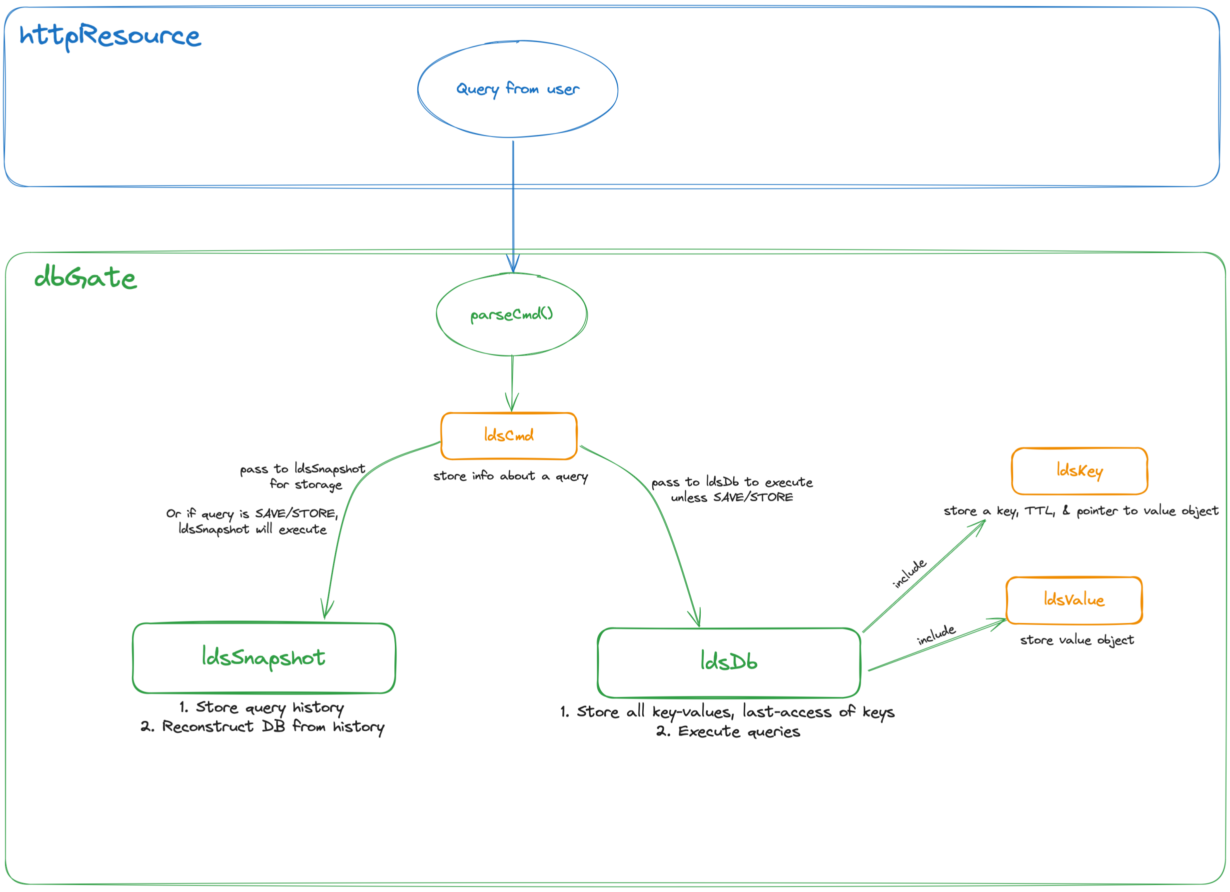
- C++ Standard Library for every utility the database offers. C++ is chosen due to its:
  - Flexibility in memory management. The database can, at any time, free memory it does not need, and allocate the exact amount of memory it needs.
  - No garbage collector. A database with garbage collector can experience frequent, in an non-deterministic fashion, slow time when the GC is working.
  - Simple concurrency control with its `std::shared_lock`, `std::unique_lock` support straight from the STL.
- Also, `httpserver` library[1] is used to create a web-server and expose one `POST /` method.

2. Client-side

- Simple Python program that sends request to database server (see `client.py`).

Overview of the data structures

- A diagram for the server data structure / execution flow:



  (rectangular entities are classes, while circular entities are functions/objects in the source code)
  ("lds" stands for "Ledis")
- The database structure is divided into 2 main classes: `ldsDb` and `ldsSnapshot`.
- `ldsDb` holds all the data: keys, and value for each key. Also executes queries from users.
- `ldsSnapshot` holds the query history: everything it needs to reconstruct the database from the bottom up upon `RESTORE` query.
  → The finer details are in the below sections.

`ldsDb`: the database

1. Data structure

- `ldsDb` maintain 2 data structures for keys and values:
  - For keys: `std::unordered_map` from C++ STL, which implemented with **hash table**.
  - For values: **linked-list** `list` for fast removal.

2. Execution

- Entry point of query execution: `ldsDb::execute(cmd)`.

- Query executions maintain that concurrent executions are conflict-serializable using **strict two-phase locking**[2] (acquire locks during execution, release locks only after the action is finished).

2.1 TTL Handling

- If an `EXPIRE key` query is issued $\rightarrow$ calculate the point in time where that `key` will expire $\rightarrow$ attach that information to the `key` (using the `ttl` field in `ldsKey` class).
- Only upon a query where `key` is accessed again, the database would examine its TTL, and if `key` is expired, remove `key` before executing the query.
  $\implies$ Not a practical approach since redundant memory of expired keys will become increasing larger. But an upside is it does not cost too much CPU resource to monitor the expiring keys.

`ldsSnapshot`: history keeper

1. Data structure

- Maintains a list of executed queries. Use **linked-list** (instead of random-access array) because inserting operations are frequent, while accessing operations are not.

2. Execution

- Only queries that modify the database are recorded.
- When `SAVE`, write out the history to binary file.
- When `RESTORE`, read query history from the binary file, and reconstruct a new database from the history.

2.1 `SAVE`

- To freeze the history in time without interrupting other query executions: **fork** the current process $\rightarrow$ from the child process, write query history to a temp file, then exit child process $\rightarrow$ parent process rename the temp file to canonical snapshot filename[3]
  $\implies$ This way, other query executions after `SAVE` are executed without delay, and by writing to a temp file first, the current snapshot will not be lost if `SAVE` has an error.

2.2 Handling `EXPIRE` queries

- `EXPIRE` queries are time-sensitive, therefore it cannot be recorded and replay later.
- Actually, `EXPIRE` queries are not recorded at all. Instead, when running `SAVE`:
  - For all keys that are expiring (have `TTL` ≥ 0), add a query `EXPIRE <KEY> <current TTL>` at the end of the history.

    - For each query in history, check if the key involves in that query still exists (or has expired). If the key does not exist anymore, ignore that query.

2.3 `RESTORE`

- Replay all queries from the snapshot file on a new, clean database, then replace current database with the new database.

Possible improvements

- `EXPIRE` query handling can definitely be improved. An approach that some databases apply can be considered: save the TTLs to a min-heap → only need to check the root TTL to see if it has expired[4].
- Conflict-serializability, while enforced using strict 2-PL, can cause dead-locks. A way to deal with this could be: detect if 2 (or more) threads are taking too long → halt the other threads that are spawned later [2-1].
- Right now, upon an execution failure, the database can be in a dirty state since there's no roll-back mechanism.

---

1. https://github.com/etr/libhttpserver↩
2. Book Fundamentals of Database Systems↩ ↩
3. Inspired by Redis snapshotting mechanism.↩
4. https://build-your-own.org/redis/13_heap↩