Concurrent KV Store

The lab I selected was the concurrent Key Value Store and the following is my design for the lab as well as the challenges I faced and the performance results.

Design

When designing the KVStore, I focused on three main concerns: use of shared_mutex, writer starvation, and snapshot consistency. The first two are related—standard shared_mutex implementations can suffer from writer starvation when continuous reads prevent writers from acquiring the lock. To resolve this, I designed a new class, FairShareMutex, which wraps a shared_mutex and mutex to implement a fair many-reader/one-writer locking scheme. It uses condition variables and counters (active_readers, active_writer, and waiting_writers) to ensure writers receive service without indefinite delay. Readers wait when writers are active or pending, guaranteeing fairness and preventing starvation.

To ensure snapshot consistency, a second FairShareMutex protects the snapshot vector within the KVStore. This prevents concurrent modifications from corrupting snapshot data, ensuring each captured snapshot remains immutable and isolated from subsequent writes.

The UpgradedKVStore introduced two optimizations to improve throughput. First, a thresholded batch write mechanism was added, where writes accumulate in a buffer until a set threshold is reached, then a background thread applies them in one lock-protected operation. This design reduced lock contention and significantly lowered latency for large write workloads, as confirmed in the batch_write benchmark—batch operations completed in roughly 200 ms compared to 800 ms for sequential individual writes, a nearly 4× speedup. Second, an upgradeable get-and-put operation was implemented, allowing threads to read a key under a shared lock and only upgrade to exclusive access when a write is needed. This reduced unnecessary blocking and improved throughput under mixed read-write workloads.

Benchmarks

1. Snapshot Consistency
The snapshot_consistency benchmark evaluates the ability of KVStore to provide consistent snapshots under concurrent writes. Multiple writer threads update overlapping keys while a separate snapshot thread periodically captures the store's state. The snapshot contents are printed, and any value that violates expected ordering is flagged as inconsistent. This design demonstrates that snapshots represent atomic, coherent views of the store even in the presence of simultaneous modifications.
2. Snapshot Isolation
The snapshot_isolation benchmark tests immutability and isolation properties of snapshots. After initializing the store with a set of values, an initial snapshot is taken, followed by a bulk update of all keys

and a second snapshot. Both snapshots are retrieved and printed, with checks to ensure the first snapshot remains unchanged and the second reflects the new state. This design confirms that updates do not retroactively affect prior snapshots, demonstrating strong snapshot isolation guarantees.

3. Snapshot Stress

The snapshot_stress benchmark stresses the store under high concurrency. Multiple threads perform tens of thousands of read and write operations while periodically taking snapshots. Sample snapshot entries and their sizes are printed to verify consistency. This design validates the store's correctness and stability under heavy read/write loads, ensuring snapshots remain accurate even when subjected to extreme stress.

4. Lock Upgrade

The lock_upgrade benchmark assesses the behavior of upgradeable locks in UpgradedKVStore. Several reader threads perform concurrent reads while multiple upgrader threads attempt to read and then update values. Each read and update is printed to confirm correctness. This design demonstrates that upgradeable locks allow concurrent reads while safely promoting to write access, preventing race conditions and ensuring consistent updates.

5. Batch Write

The batch_write benchmark measures performance differences between batch insertion and sequential individual writes. A large set of key-value pairs is written using put_batch and then individually, with the execution time for each method recorded. This design confirms both correctness and efficiency of batch operations, showing that bulk writes are functionally equivalent to individual writes but significantly faster.

Challenges

Getting CMake to work properly for this project was also a surprisingly challenging part of the process. Because the project involved multiple source directories, shared headers, and separate build targets for each benchmark, setting up the correct include paths, output directories, and dependency rules required several iterations. Linking the FairShareMutex, KVStore, and upgraded KVStore components under one unified build system exposed subtle issues—like missing header paths, mismatched target names, and race conditions in parallel builds. Debugging these problems helped me understand how CMake handles target-level dependencies and object files, and by the end, I had a clean, modular build setup that could scale as new benchmarks and implementations were added.

Implementing batch writes was another major challenge. The goal was to improve performance by aggregating write operations, but designing it to be both thread-safe and non-blocking required careful synchronization. It wasn't enough to simply push writes into a queue; I needed a dedicated background thread that would process writes once a threshold was reached, without blocking ongoing reads or interfering with snapshot consistency. Getting the timing and signaling right with condition variables took several attempts—especially ensuring that the worker thread didn't deadlock or miss notifications when shutting down. After multiple iterations and testing with large workloads, the batch write system worked reliably, showing clear performance gains while maintaining correctness under concurrency.