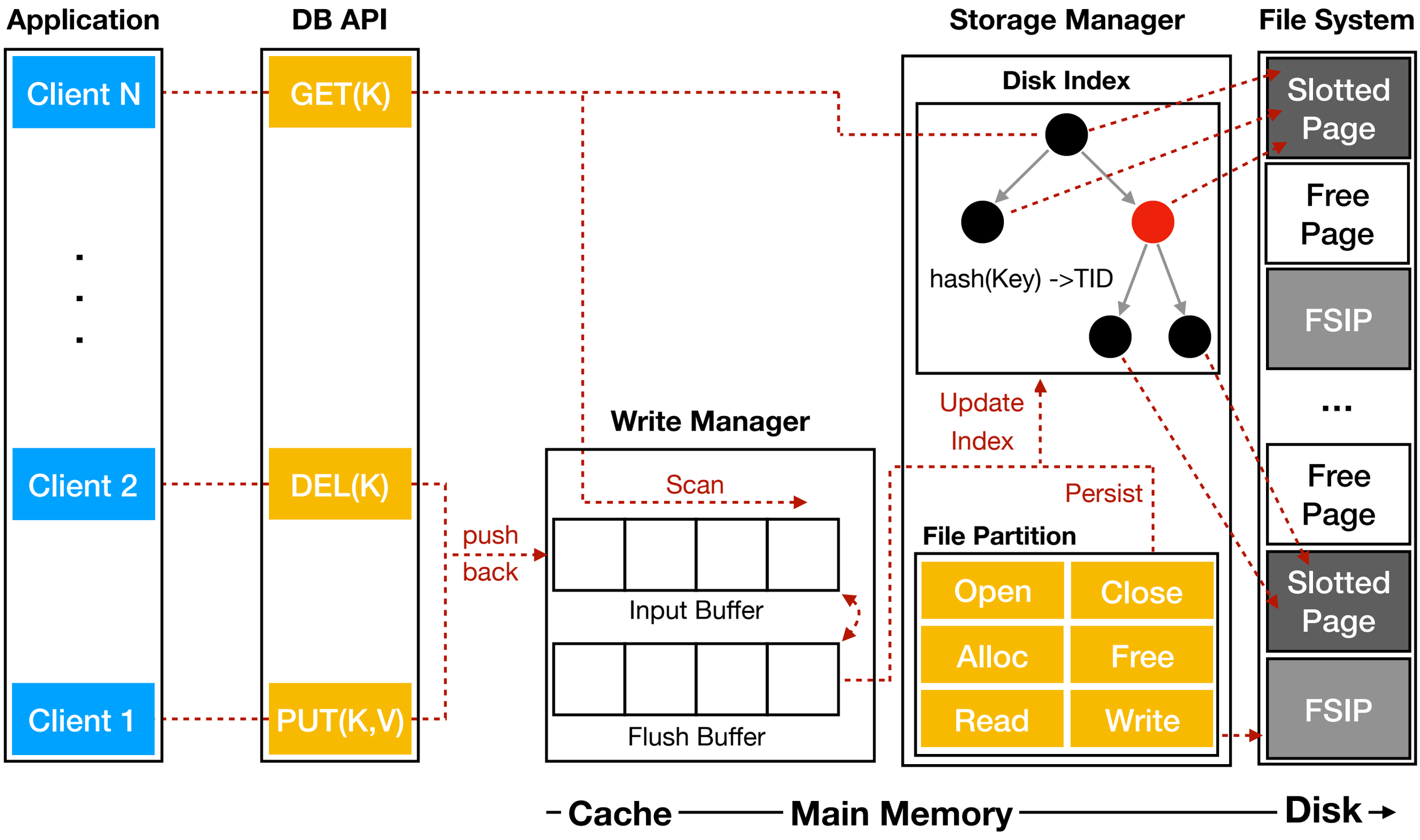
keyDB - An Overview

Design & Implementation of a simple Key-Value Store

Technology Overview

- **Compiler**: gcc 8.2
- **Build Process**: CMake
- **External Libraries**: Boost Asio, OpenSSL
- **Test Framework**: Catch2

Architecture Overview



Architecture Overview (Remarks)

- Writes are collected in-memory and are written in a sequential batch process

- This utilizes characteristics of both HDDs and SSDs

- Updates and deletes are performed lazily

- Parallel writes/reads are synchronized using locks

- The memory budget currently only limits the size of the write buffer and thus more memory is actually used

Performance Characteristics (Writes)

- Inserts, Updates and Deletes always into in-memory buffer.
- Writes to the buffer are append only: O(1) runtime
- Parallel writes to the buffer are synchronized using locks
- Buffer grows up to a given memory budget *B*: O(*B*) space
- Upon reaching the memory limit, the input buffer is swapped with an empty flush buffer.
- Incoming writes are redirected to the empty buffer while the full buffer is flushed to secondary storage

Performance Characteristics (Flush)

- A flusher thread takes care of the full buffer
- The key-value records in the full buffer are written sequentially in batches to slotted pages: O(*B*) runtime
- In parallel, while the data is written to secondary storage, an index data structure (BST) is constructed: O(n) space
- This index structure maps from hashed key values to its TID (page number and slot number)
- Throughout this process the index and slotted pages are locked and can not be accessed by other queries

Performance Characteristics (Reads)

- Read queries first scan the in-memory buffer. If a valid or deleted key is found, the query can return without disk I/O
- Otherwise, the key is searched (O(log n)) within the disk index. If it is not found, the query returns without disk I/O
- If the key is found in the index, its TID is used for fetching the correct page. Next, the key-value record is returned

Cache Strategy

- This simple key-value store was designed with the assumption of much more writes than reads and very limited memory resources

- Recently written records are evicted as soon as the memory buffer reaches the memory limit

- As there is no read cache implemented, no cache/eviction strategy (in the usual sense) is employed

Scaling

- With many parallel queries, runtime performance will drop significantly because of lock contention

- Right now, as there is no smart caching employed, the benefit of a higher memory budget is limited

- The system scales well with increasing data set size

- By not storing the string keys in the disk index, the memory footprint of the index is significantly reduced

Improvements

- Using a proper LSM provides indexed access to the write buffer
- Use of cache conscious data structures (e.g., CSB+ tree)
- For keys with the same hash value, prefetch the pages
- Employ garbage collection process to reuse the occupied space by (soft) deleted and old records
- *Hot data* read cache with ARC or 2nd chance replacement
  - E.g.: https://github.com/WeberNick/masterDB/tree/master/src/buffer
- Employ compression
- …