System Design Document: High-Performance Transactional KV Store

1. Introduction

This document outlines the system design for a high-performance, single-node, transactional key-value storage engine. The system is engineered to provide full ACID compliance through a lock-free Multi-Version Concurrency Control (MVCC) model, ensuring ultra-fast reads and writes. The architecture combines a Log-Structured Merge-Tree (LSM-Tree) for the data path with a persistent Adaptive Radix Tree (ART) for primary indexing.

2. System Architecture Overview

The engine is composed of several distinct but interconnected components. The design philosophy is to separate concerns, allowing each component to be optimized for its specific task.

Architectural Diagram (Logical Flow):

```
Client Request -> Transaction Manager
|
    +-- Write Path: WAL -> Memtable -> Persistent ART (Path Copy)
|
    +-- Read Path:  Transaction Snapshot -> Memtable -> Persistent ART -> Bloom Filter -> Block Cache -> SSTable
```

```
|
    +-- Commit:    WAL (Commit Record) -> Atomic State Update

Background Threads:
|
    +-- Flusher:   Immutable Memtable -> SSTable (Level-0)
|
    +-- Compactor:  SSTable (Level N) + SSTable (Level N+1) -> New SSTable (Level N+1) -> MANIFEST Update
```

3. Component Design

3.1. I/O and Persistence Layer

This layer is responsible for all interactions with the physical storage medium.

- **Write-Ahead Log (WAL):** The cornerstone of durability and atomicity.[1]
  - **Function:** Records every database modification (Put, Delete, Commit) before it is applied elsewhere. Enables recovery after a crash by replaying the log.[3]
  - **On-Disk Format:** A sequence of binary records. Each record is prefixed with a [length (4 bytes), crc32 (4 bytes)] header, followed by the serialized payload (TxnID, RecordType, Key, Value).[4]
  - **Mechanism:** Implements **group commit** to amortize the cost of fsync() across multiple transactions, dramatically increasing write throughput.[1]
- **SSTable (Sorted String Table):** Immutable on-disk files that store the bulk of the data.[5]
  - **Function:** Persistently stores key-value pairs sorted by key.
  - **On-Disk Format:** A block-based structure for efficient access.[6]
    1. **Data Blocks:** Contain sorted, prefix-compressed key-value pairs.
    2. **Filter Block:** A Bloom filter built over all keys in the SSTable.
    3. **Index Block:** An index of data blocks, mapping divider keys to block offsets.
    4. **Footer:** A fixed-size block at the end containing pointers to the filter and index blocks.
  - **Serialization:** Uses an efficient binary format like Protocol Buffers or MessagePack to minimize space and parsing overhead.[7]
- **Bloom Filter:** A probabilistic data structure for optimizing reads.[9]
  - **Function:** Quickly tests for the absence of a key in an SSTable, avoiding unnecessary

disk I/O.[11]
    - **Integration:** One Bloom filter is generated per SSTable and stored in its Filter Block. It is loaded into memory when the SSTable is active.
- **MANIFEST File:** The source of truth for the database's on-disk state.[13]
    - **Function:** An append-only log that records every change to the LSM-Tree's structure (e.g., SSTable creation, deletion).
    - **Mechanism:** State transitions are made atomic by writing a new MANIFEST file and then using an atomic rename() system call to update a CURRENT file pointer.

3.2. In-Memory Components

This layer manages data and structures held in RAM for performance.

- **Memtable:** An in-memory write buffer.[6]
    - **Function:** Absorbs recent writes at memory speed, deferring disk I/O.
    - **Data Structure:** A **lock-free Skip List** implemented with std::atomic. This structure provides O(logn) complexity for operations and naturally keeps keys sorted, which is essential for flushing to SSTables.[6]
    - **Mechanism:** Implements **pipelining**. When the active Memtable is full, it is atomically swapped with a new empty one and placed in a queue for a background thread to flush, preventing write stalls.[15]
- **Block Cache:** An in-memory cache for SSTable data blocks.
    - **Function:** Reduces read latency by storing frequently accessed data blocks in RAM, avoiding disk reads.
    - **Mechanism:** Implements a standard LRU (Least Recently Used) eviction policy.

3.3. Indexing Layer

- **Persistent Adaptive Radix Tree (ART):** The primary index for the entire database.[16]
    - **Function:** Maps every key to its version chain. Its performance is dependent on key length, not the number of keys, making it ideal for point lookups.
    - **Persistence Mechanism:**
        1. **On-Disk Format:** Serialized using a **post-order traversal**, which stores child disk locations within the parent node's on-disk representation.[16] This is the key to enabling lazy loading.
        2. **Lazy Loading & Pointer Swizzling:** To keep in-memory nodes compact, a single 64-bit pointer is used for both memory addresses and disk locations. The most

significant bit acts as a "swizzle flag" to differentiate between an in-memory pointer and an on-disk location, enabling on-demand loading of index nodes.[16]

3.4. Concurrency and Transaction Layer

This layer orchestrates transactional operations and ensures ACID properties.

- **Transaction Manager:** The central coordinator and public API.
  - **Function:** Manages the lifecycle of transactions.
  - **Mechanisms:**
    - **TxnID Generation:** Uses a global std::atomic<uint64_t> to generate unique, monotonically increasing transaction IDs with a lock-free fetch_add.[18]
    - **Snapshot Management:** Creates a consistent snapshot for each transaction when it begins.
    - **Commit Protocol:** Coordinates the two-phase commit logic (writing to WAL, then making changes visible).
- **MVCC (Multi-Version Concurrency Control):** The core of the lock-free concurrency model.[20]
  - **Function:** Allows concurrent reads and writes by maintaining multiple versions of data items.
  - **Mechanisms:**
    1. **Versioned Records:** Each key-value pair is stored as (key, creation_TxnID, deletion_TxnID, value).[21] A delete operation creates a new version with a "tombstone" marker.[23]
    2. **Transaction Snapshot:** Defined by the transaction's own TxnID and a list of all other transactions that were in-progress when it started.[21]
    3. **Visibility Rule:** A version is visible to a transaction if its creation_TxnID is committed and older than the transaction's snapshot, and its deletion_TxnID is not.[21]
    4. **Path Copying:** The Persistent ART is updated using path copying. A write creates a new version of the tree by copying only the nodes on the path to the modified leaf. This leaves the old tree intact for existing snapshots.[17]

3.5. Background Processes

These processes are essential for long-term system health and performance.

- **Compactor:** A pool of background threads for merging SSTables.[15]
  - **Function:** Reclaims space from deleted/obsolete data, reduces read amplification, and maintains the LSM-Tree structure.
  - **Strategy:** Implements **Leveled Compaction**, which provides a good balance of read, write, and space amplification for mixed workloads.[25] It merges a file from Level-L with overlapping files in Level-(L+1).
- **Garbage Collector:** Integrated with the compaction process to reclaim obsolete versions.[27]
  - **Function:** Physically removes data and index versions that are no longer visible to any active transaction.
  - **Mechanism:** The Transaction Manager determines a "low-water mark" TxnID (the ID of the oldest active transaction). During compaction, any version that was deleted by a transaction committed before this mark is discarded and not written to the new SSTable.[27] Obsolete ART nodes are reclaimed via a similar mark-and-sweep process.[29]

4. Component Integration and Data Flow

The components work in concert to process database operations.

- **Write Operation:** A Put(key, value) call within a transaction is routed to the TransactionManager. It is first written durably to the WAL. Then, a versioned entry is inserted into the lock-free Memtable (Skip List). Concurrently, a new version of the Persistent ART is created via path copying, private to the transaction.
- **Read Operation:** A Get(key) call uses the transaction's snapshot, which includes a pointer to a specific ART root. The read checks the Memtable first. If not found, it traverses the Persistent ART to find the location of the key's version chain on disk. Before reading an SSTable, its Bloom Filter is checked to avoid the I/O if the key is absent. If present, the relevant data block is read from the SSTable (via the Block Cache) and the visibility rule is applied to the version chain to find the correct data.
- **Commit Operation:** The TransactionManager writes a COMMIT_TX record to the WAL and ensures it is flushed to disk. This is the atomic commit point. After the flush, a global atomic pointer to the latest committed Persistent ART root is updated to point to the new root created by the transaction, making its changes visible to all subsequent transactions.

Implementation Plan

This plan follows a phased, bottom-up approach, prioritizing the creation of independently testable components before integrating them into the complete system.

Phase 1: Core Data Structures & I/O Primitives

Goal: Build the fundamental, self-contained building blocks of the engine. All components in this phase can be developed and unit-tested in isolation.

1. **Module: Low-Level I/O Abstraction**
    - **Task:** Create a C++ StorageManager class that abstracts platform-specific file operations (open, read, write, fsync, rename).
    - **Testing:** Mock the file system interface to test the wrapper's logic without actual disk I/O.
2. **Module: On-Disk Data Formats**
    - **Task:** Implement the serialization and deserialization logic for all custom file formats.
    - **Sub-tasks:**
        - WAL Record: Create a class with serialize() and deserialize() methods, including CRC32 checksum calculation and verification.[4]
        - SSTable Blocks: Implement classes for Data, Index, and Filter blocks, and the SSTable Footer.
    - **Testing:** Write extensive unit tests to ensure that a serialized object can be deserialized back into its original state perfectly. Test for corruption detection using the CRC.
3. **Module: Standalone Data Structures**
    - **Task:** Implement the core in-memory and probabilistic data structures.
    - **Sub-tasks:**
        - BloomFilter: A standard implementation with add() and might_contain() methods.
        - SkipList: A single-threaded, sorted Skip List implementation.
        - Adaptive Radix Tree (ART): A single-threaded, in-memory implementation of ART with adaptive node types.[17]
    - **Testing:** Unit test each data structure for correctness on all operations (insert, delete, lookup, iteration).

Phase 2: Building the Write & Read Path

Goal: Assemble the Phase 1 components into functional read and write data paths.

1. **Module: WAL Manager**
    - **Dependencies:** Low-Level I/O, WAL Record Format.
    - **Task:** Create a WALManager class responsible for appending records and replaying a log file for recovery. Implement group commit logic.
    - **Testing:** Test append correctness. Simulate a crash by terminating the process and verify that the WALManager can successfully replay the log to reconstruct state.
2. **Module: SSTable Manager**
    - **Dependencies:** Low-Level I/O, SSTable Formats, Bloom Filter.
    - **Task:**
        - SSTableBuilder: A class that takes a stream of sorted key-value pairs and writes a valid SSTable file.
        - SSTableReader: A class that opens an SSTable, loads its index and filter, and provides a Get(key) method.
    - **Testing:** Create SSTables with the builder and verify their contents with the reader. Test the Bloom filter's effectiveness in skipping reads.
3. **Module: Memtable & Flush Logic**
    - **Dependencies:** Skip List, SSTable Manager.
    - **Task:** Wrap the Skip List in a Memtable class. Implement the logic to flush its contents to a new Level-0 SSTable using the SSTableBuilder.
    - **Testing:** Populate a Memtable, flush it, and use an SSTableReader to verify that all data was written correctly and in sorted order.

Phase 3: Concurrency and Transactions

Goal: Introduce the MVCC framework and make the data paths thread-safe and transactional.

1. **Module: Lock-Free Data Structures**
    - **Dependencies:** Skip List, ART.
    - **Task:** Convert the single-threaded SkipList and ART from Phase 1 into concurrent, lock-free versions using std::atomic and compare-and-swap (CAS) loops.[31]
    - **Testing:** Write rigorous multi-threaded tests to hammer the data structures with concurrent reads, writes, and deletes to check for race conditions and correctness.
2. **Module: MVCC Core**
    - **Task:**
        - Implement the TransactionManager for issuing TxnIDs and managing transaction

states.
        - Modify the Memtable and SSTable formats to handle versioned records (creation_TxnID, deletion_TxnID).
        - Implement the snapshot creation and visibility check logic.
    - **Testing:** Write unit tests for the visibility rule. Create multi-threaded tests to verify transaction isolation (e.g., a reader transaction should not see data from a concurrent, uncommitted writer transaction).
3. **Module: Persistent & Versioned ART**
    - **Dependencies:** Concurrent ART.
    - **Task:**
        - Implement path copying on the concurrent ART to support non-destructive updates.[24]
        - Implement the on-disk serialization (post-order traversal) and lazy-loading (pointer swizzling) mechanisms.[16]
    - **Testing:** Test that updates create new tree versions while leaving old ones accessible. Test persistence by writing an ART to a file, restarting, and lazy-loading nodes to satisfy a query.

Phase 4: Background Processes & Final Integration

Goal: Implement the background maintenance tasks and assemble the final, user-facing database engine.

1. **Module: Compaction Engine**
    - **Dependencies:** SSTable Manager, MANIFEST format.
    - **Task:** Implement a background thread pool for compaction. Code the Leveled Compaction strategy logic for selecting, merging, and replacing SSTables, and for updating the MANIFEST file atomically.
    - **Testing:** Create a multi-level LSM-Tree structure on disk. Trigger a compaction and verify that files are correctly merged, old files are deleted, and the MANIFEST is updated to a new valid state.
2. **Module: Garbage Collection**
    - **Dependencies:** Compaction Engine, Transaction Manager.
    - **Task:** Integrate GC logic into the compaction process. The TransactionManager must expose the low-water mark. The compactor must use this to identify and discard obsolete versions during a merge.
    - **Testing:** Run a workload with updates and deletes. Start a long-running read transaction to hold the low-water mark. Run compaction and verify that old versions are retained. Commit the read transaction, run compaction again, and verify that the old versions are now purged.

3. **Module: Top-Level API Integration**
    - **Dependencies:** All other components.
    - **Task:** Create the final Database class that encapsulates all components and exposes the public API (Open, Put, Get, Delete, BeginTransaction).
    - **Testing:** Write end-to-end integration tests that simulate realistic application workloads, including concurrent transactions, high write volumes to trigger flushing and compaction, and crash-recovery scenarios.

Works cited

1. Writing A Database: Part 2 — Write Ahead Log | by Daniel Chia | Medium, accessed September 4, 2025, https://medium.com/@daniel.chia/writing-a-database-part-2-write-ahead-log-2463f5cec67a
2. Understanding Write-Ahead Logs in Distributed Systems | by Abhishek Gupta | Medium, accessed September 4, 2025, https://medium.com/@abhi18632/understanding-write-ahead-logs-in-distributed-systems-3b36892fa3ba
3. How RocksDB works - Artem Krylysov, accessed September 4, 2025, https://artem.krylysov.com/blog/2023/04/19/how-rocksdb-works/
4. LevelDB Explained - How To Read and Write WAL Logs, accessed September 4, 2025, https://selfboot.cn/en/2024/08/14/leveldb_source_wal_log/
5. LevelDB - Database of Databases, accessed September 4, 2025, https://dbdb.io/db/leveldb
6. Building an LSM-Tree Storage Engine from Scratch - DEV Community, accessed September 4, 2025, https://dev.to/justlorain/building-an-lsm-tree-storage-engine-from-scratch-3eom
7. Best practices for custom file structures - Stack Overflow, accessed September 4, 2025, https://stackoverflow.com/questions/600708/best-practices-for-custom-file-structures
8. The Architect's Guide to Data and File Formats - MinIO Blog, accessed September 4, 2025, https://blog.min.io/architects-guide-data-file-formats/
9. Bloom filter | Docs - Redis, accessed September 4, 2025, https://redis.io/docs/latest/develop/data-types/probabilistic/bloom-filter/
10. Bloom filter - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Bloom_filter
11. The Fundamentals of RocksDB - Stream, accessed September 4, 2025, https://getstream.io/blog/rocksdb-fundamentals/
12. Starskey - Fast Persistent Embedded Key-Value Store (Inspired by LevelDB) : r/golang, accessed September 4, 2025, https://www.reddit.com/r/golang/comments/1i5q2ps/starskey_fast_persistent_embedded_keyvalue_store/
13. LevelDB (go version) architecture design analysis - go学习 - SegmentFault 思否,

accessed September 4, 2025, https://segmentfault.com/a/1190000040286395/en
14. The Log-Structured Merge-Tree (LSM-Tree) - UMass Boston CS, accessed September 4, 2025, https://www.cs.umb.edu/~poneil/lsmtree.pdf
15. RocksDB Overview · facebook/rocksdb Wiki · GitHub, accessed September 4, 2025, https://github.com/facebook/rocksdb/wiki/RocksDB-Overview
16. Persistent Storage of Adaptive Radix Trees in DuckDB – DuckDB, accessed September 4, 2025, https://duckdb.org/2022/07/27/art-storage.html
17. VART: A Persistent Data Structure For Snapshot Isolation - SurrealDB, accessed September 4, 2025, https://surrealdb.com/blog/vart-a-persistent-data-structure-for-snapshot-isolation
18. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control - VLDB Endowment, accessed September 4, 2025, https://www.vldb.org/pvldb/vol10/p781-Wu.pdf
19. Lock-free Serializable Transactions - UT Computer Science, accessed September 4, 2025, https://www.cs.utexas.edu/ftp/techreports/tr05-04.pdf
20. Multiversion concurrency control - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Multiversion_concurrency_control
21. Implementation of MVCC Transactions for Key-Value Stores - Highly Scalable Blog, accessed September 4, 2025, https://highlyscalable.wordpress.com/2012/01/07/mvcc-transactions-key-value/
22. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems, accessed September 4, 2025, https://15721.courses.cs.cmu.edu/spring2016/papers/p677-neumann.pdf
23. Persistent Trie | Set 1 (Introduction) - GeeksforGeeks, accessed September 4, 2025, https://www.geeksforgeeks.org/dsa/persistent-trie-set-1-introduction/
24. How MVCC databases work internally | by Kousik Nath - Medium, accessed September 4, 2025, https://kousiknath.medium.com/how-mvcc-databases-work-internally-84a27a380283
25. Size Tiered and Leveled Compaction Strategies STCS + LCS - ScyllaDB University, accessed September 4, 2025, https://university.scylladb.com/courses/scylla-operations/lessons/compaction-strategies/topic/size-tiered-and-leveled-compaction-strategies-stcs-lcs/
26. Universal Compaction · facebook/rocksdb Wiki - GitHub, accessed September 4, 2025, https://github.com/facebook/rocksdb/wiki/universal-compaction
27. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA, accessed September 4, 2025, https://15721.courses.cs.cmu.edu/spring2019/papers/05-mvcc3/p1307-lee.pdf
28. MVCC garbage collection - TiDB Development Guide, accessed September 4, 2025, https://pingcap.github.io/tidb-dev-guide/understand-tidb/mvcc-garbage-collection.html
29. Garbage Collection Technique for Non-volatile Memory by Using Tree Data Structure, accessed September 4, 2025,

https://www.researchgate.net/publication/298727199_Garbage_Collection_Technique_for_Non-volatile_Memory_by_Using_Tree_Data_Structure
30. en.wikipedia.org, accessed September 4, 2025, https://en.wikipedia.org/wiki/Persistent_data_structure#:~:text=Because%20persistent%20data%20structures%20are,counting%20or%20mark%20and%20sweep.
31. A Lock-Free Stack: A Simplified Implementation – MC++ BLOG - Modernes C++, accessed September 4, 2025, https://www.modernescpp.com/index.php/a-lock-free-stack-a-simplified-implementation/