Architecting a High-Performance, Lock-Free Transactional Key-Value Store in C++

Part I: Foundational Architecture

This report delineates the architectural design and implementation strategy for a high-performance, single-node, transactional key-value storage engine engineered in C++. The design prioritizes ultra-fast, concurrent read and write operations, full ACID compliance, and a lock-free concurrency model. This is achieved through a synergistic combination of a Log-Structured Merge-Tree (LSM-Tree) for the physical data path and a persistent, version-aware Adaptive Radix Trie (ART) for primary indexing. The document provides a comprehensive blueprint, covering the system's foundational components, its novel indexing and concurrency control mechanisms, and the operational procedures required for long-term maintenance and performance stability.

Section 1: Architectural Philosophy and Core Components

The architectural philosophy is rooted in minimizing contention on the critical read and write paths by embracing immutability and atomic state transitions. Instead of relying on traditional locking mechanisms that serialize access, this design leverages lock-free data structures and Multi-Version Concurrency Control (MVCC) to enable high degrees of parallelism.

1.1. A Hybrid Architectural Model

The storage engine is architected around a hybrid model that integrates the write-path efficiency of a Log-Structured Merge-Tree (LSM-Tree) with the lookup performance of a persistent, version-aware Trie index. This approach is deliberately chosen to capitalize on the distinct advantages of each structure, creating a system optimized for a balanced workload of both rapid writes and low-latency reads.[1]

LSM-Tree architectures, famously implemented in systems like LevelDB and RocksDB, are exceptionally well-suited for write-intensive workloads. They achieve this by transforming random write operations into highly efficient, sequential, append-only writes to disk.[4] This design minimizes costly disk seeks and maximizes throughput. However, in a conventional LSM-Tree, a read operation may need to check multiple on-disk files (SSTables) across various levels of the tree to find the correct version of a key, a process that can introduce latency.

To mitigate this read-path overhead, this architecture decouples primary indexing from the physical data layout. Instead of relying solely on SSTable-internal indexes and Bloom filters, the engine employs a centralized, persistent Trie as the canonical index. A Trie, specifically an Adaptive Radix Tree (ART), provides a highly efficient lookup path with a complexity dependent on key length rather than the total number of keys, making it ideal for point and prefix queries.[7] By making this Trie persistent and version-aware, it can serve as the definitive entry point for all read operations, directly guiding a query to the precise location of the required data version, whether in memory or on disk.

The system is composed of several interacting components, each designed with concurrency and performance in mind:

- **Transaction Manager:** This is the public-facing API and the orchestrator of all database operations. It is responsible for managing the lifecycle of transactions, assigning unique transaction identifiers, creating transaction snapshots for isolation, and coordinating the commit protocol.
- **Write-Ahead Log (WAL):** A strictly append-only log file that provides durability and atomicity. All data modifications are first recorded in the WAL before being applied to any other part of the system.[5]
- **Memtable:** An in-memory, concurrent, sorted data structure that acts as a write buffer. It absorbs recent writes at memory speed, deferring their persistence to the main on-disk storage structures.[1]
- **Persistent Trie Index:** The primary index for the entire database. It maintains a versioned mapping of every key to its corresponding data location. Its persistent nature ensures that snapshots can be maintained for MVCC, and its structure allows for rapid lookups.
- **SSTable Manager:** This component is responsible for creating, managing, and accessing the on-disk Sorted String Table (SSTable) files, which are immutable files containing the bulk of the user data.
- **Compactor:** A set of background threads responsible for the crucial maintenance tasks of merging SSTables to reclaim space, remove obsolete data versions, and maintain the

structural health of the LSM-Tree.[12]

1.2. Data Flow Analysis

The interaction of these components defines the primary data flows through the engine.

- **Write Path:** A Put(key, value) operation initiated within a transaction follows a strict sequence to ensure ACID properties. First, a log record detailing the operation, including the transaction ID, key, and value, is appended to the Write-Ahead Log file. This log entry is then flushed to the physical storage medium to guarantee durability. Once durable, the same key-value pair, augmented with its versioning metadata (i.e., its creation transaction ID), is inserted into the active in-memory Memtable. Simultaneously, the Persistent Trie Index is updated using a lock-free "path copying" technique, which creates a new logical version of the index reflecting the write. This new index version remains private to the transaction until it successfully commits.
- **Read Path:** A Get(key) operation is executed within the context of a transaction's consistent snapshot. The read begins by consulting the version of the Trie Index associated with that snapshot. The search first probes the active Memtable, which contains the most recent, un-flushed data. If the key is not found in the Memtable, the Trie Index is traversed to identify the specific SSTable on disk that contains the relevant version of the key.[1] Before performing any disk I/O, the engine consults a Bloom filter associated with the target SSTable. This probabilistic check can definitively rule out the key's presence in the file, thus avoiding an expensive and unnecessary disk read.[14] If the Bloom filter indicates a probable match, the engine reads the relevant data block from the SSTable and locates the key.
- **Commit Path:** The commit process finalizes a transaction's changes. A "commit" record, containing the transaction ID, is written to the WAL and durably persisted. This action serves as the atomic commit point. Once the commit record is on disk, the transaction's modifications, which are already reflected in the Memtable and a new logical version of the Trie, are made globally visible. This is accomplished by atomically updating a global pointer to designate the transaction's new Trie root as the latest committed version of the index. Subsequent transactions will then start with a snapshot based on this newly committed state.

A central design tenet of this architecture is the careful application of lock-free principles, which extends beyond individual data structures to the management of the entire system's state. The "no-lock" requirement is interpreted not as the absolute absence of any mutual exclusion primitive, but as a commitment to ensuring the primary read and write paths are non-blocking. This is achieved by architecting the system around atomic state transitions that operate on immutable data structures.

Traditional database designs often rely on locks to protect shared data during modification, which can lead to contention and limit scalability. In this engine, core components of the database state—such as the active Memtable, the collection of on-disk SSTables, and the root of the Trie index—are treated as immutable snapshots. When a change is required, the system does not modify these structures in-place. Instead, it creates new versions. For example, when a Memtable becomes full, it is not locked for flushing. It is atomically swapped with a new, empty Memtable via a single atomic pointer exchange, a classic lock-free programming pattern.[16] The now-immutable old Memtable can be safely flushed to disk by a background thread without blocking new incoming writes.

Similarly, the compaction process does not alter existing SSTables. It reads a set of immutable SSTables and produces a new set of SSTables as its output. The transition from the old set of files to the new one is recorded as an atomic update to a metadata file (the MANIFEST). This ensures that readers always see a consistent view of the database files. The Persistent Trie Index follows the same principle through path copying, where updates generate a new root pointer. The entire system can thus be viewed as a sequence of states, with transitions between states accomplished via atomic operations on shared pointers. This design provides lock-free progress for concurrent readers and writers on the hot path, relegating more complex synchronization to background maintenance tasks and the commit protocol, thereby maximizing throughput and scalability on multi-core hardware.

Section 2: The Write Path: Durability and In-Memory Buffering

The write path is meticulously engineered for maximum throughput and durability. It leverages a Write-Ahead Log to guarantee data safety and an in-memory Memtable to buffer and sort incoming writes, effectively decoupling the client-facing write latency from the inherent latency of disk I/O.

2.1. The Write-Ahead Log (WAL): The Cornerstone of Durability

The Write-Ahead Log is the foundational component for ensuring the 'A' (Atomicity) and 'D' (Durability) of ACID transactions. The fundamental rule of WAL is that any modification to the database state must be recorded and persisted in the log *before* the change is applied elsewhere and before the transaction is acknowledged as committed to the client.[10] This protocol guarantees that in the event of a system crash, the engine can be restored to a consistent state by replaying the log. Any committed transactions whose changes were not yet fully persisted in the main data structures can be redone from their log records, while any incomplete transactions will be implicitly rolled back as they lack a commit record in the log.[18]

The on-disk format of the WAL is a custom binary protocol designed for efficiency and robustness. Each log record is structured as a self-contained unit, prefixed with a 32-bit integer specifying the payload length and a 32-bit CRC checksum of the payload to detect corruption or partial writes.[10] The payload itself is a serialized structure containing:

- **Transaction ID (TxnID):** A 64-bit unique identifier for the transaction.
- **Record Type:** An enumeration indicating the nature of the operation (e.g., BEGIN_TX, PUT, DELETE, COMMIT_TX, ABORT_TX).
- **Key Length and Key Data:** The key being modified.
- **Value Length and Value Data:** The value associated with a PUT operation.

This format, inspired by designs in LevelDB and etcd, allows for efficient parsing during recovery.[10]

A significant performance challenge with any WAL implementation is the latency imposed by the fsync() system call (or its equivalent), which is required to ensure data is physically written to the storage medium rather than just sitting in the operating system's buffer cache. A naive implementation that calls fsync() for every transaction commit would severely limit write throughput.[10] To overcome this, the engine will implement

group commit. When multiple transactions are committing concurrently, their commit records are buffered in memory. A single writer thread then writes this batch of records to the WAL file in one I/O operation and follows it with a single fsync() call. This amortizes the high cost of the disk sync across many transactions, dramatically increasing the sustainable commit rate and overall write throughput of the system.[17]

The recovery process upon engine startup is straightforward and robust. The engine reads the WAL sequentially from the last known checkpoint. It reconstructs the in-memory state (the Memtable and a view of the Trie index) by replaying the operations from all transactions that have a COMMIT_TX record. If a transaction has log entries but lacks a corresponding COMMIT_TX record, it is considered to have been in-flight during the crash; its operations are discarded, effectively rolling it back.[5] Once recovery is complete, the engine is in a state consistent with all successfully committed transactions.

2.2. The Memtable: A Lock-Free In-Memory Skip List

The Memtable serves as the in-memory write buffer, absorbing recent modifications at RAM speed. The choice of data structure for the Memtable is critical for both write performance and the efficiency of subsequent flushing to disk. A Skip List, as employed by LevelDB and RocksDB, is the ideal candidate.[1] It offers an average complexity of

O(logn) for search, insertion, and deletion operations, similar to a balanced binary tree. However, its probabilistic nature and simpler structural rules make it significantly more amenable to a highly concurrent, lock-free implementation.[2] A key advantage of the Skip List is that it naturally maintains its elements in sorted key order, which is a prerequisite for efficiently generating the sorted SSTable files when the Memtable is flushed to disk.

The Skip List will be implemented using C++ std::atomic templates to manage its forward pointers at every level. This allows for lock-free modifications. An insertion operation, for example, proceeds by first locating the correct insertion points at each level of the Skip List. It then constructs a new node. Finally, it uses atomic compare-and-swap (CAS) operations within a loop to atomically link the new node into the list at each level. If a CAS operation fails because another thread modified the list concurrently, the operation is simply retried on that level. This optimistic approach allows multiple writer threads to modify the Skip List simultaneously without ever acquiring a mutex, and it permits reader threads to traverse the list without being blocked by writers.[16]

To prevent write operations from stalling when the active Memtable fills up, the engine will implement **Memtable pipelining**, a technique refined in RocksDB.[21] The system maintains a pointer to an

active Memtable and a queue of *immutable* Memtables. All new writes are directed to the active Memtable. When its size exceeds a configured threshold, two actions occur in a single atomic operation: the current active Memtable is moved to a queue of immutable Memtables awaiting flushing, and a new, empty Memtable is designated as the active one. This transition is managed by an std::atomic pointer, ensuring it is instantaneous and thread-safe. A dedicated pool of background threads services this queue, taking immutable Memtables one by one and writing their sorted contents to new SSTable files on disk. This architecture effectively decouples the client-facing write path from the slower disk I/O path, allowing the engine to sustain a high write rate even when the storage backend is temporarily slow.

Section 3: On-Disk Storage and Data Layout

The on-disk storage subsystem is designed for durability, compactness, and efficient retrieval. It is centered around immutable SSTable files for data storage, augmented by Bloom filters for read optimization, and a MANIFEST file to atomically track the database's state.

3.1. The SSTable (Sorted String Table) File Format

SSTables are immutable, ordered files that serve as the primary persistent store for the database's key-value data. The design of our custom SSTable format is heavily influenced by the proven, block-based structure of LevelDB and RocksDB, which is optimized for efficient scanning and random access.[1] A single SSTable file is logically partitioned into several sections:

- **Data Blocks:** These are the fundamental units of data storage, typically 4 KB to 64 KB in size. Each data block contains a sequence of key-value pairs, sorted by key. To reduce storage footprint, keys within a block can be prefix-compressed. For example, instead of storing "apple", "apply", "apricot", the engine would store "apple", "(4)y", "(2)ricot", where the number indicates the shared prefix length.
- **Filter Block:** For each SSTable, a corresponding Bloom filter is constructed over all the keys contained within it.[13] This filter is stored as a distinct block within the SSTable file. Its purpose is to accelerate lookups for keys that do not exist in the file, as will be detailed below.
- **Index Block:** This block serves as a directory for the data blocks. It contains a sorted list of key-value pairs where the key is a "divider" key (e.g., the first key of each data block) and the value is a pointer (file offset and size) to that data block. To find a specific key, a reader can perform a binary search on the index block to quickly identify which data block *might* contain the key, thus avoiding a full scan of the SSTable.
- **Footer:** This is a fixed-size section at the very end of the SSTable file. Its fixed position allows it to be read easily. The footer contains essential metadata, including a magic number to identify the file type and version, and pointers (file offsets) to the starting locations of the Filter Block and the Index Block.

For serialization of data within these blocks, a structured binary format is paramount for performance and compactness. While a custom format can be developed, leveraging existing, well-vetted libraries such as Google's Protocol Buffers or MessagePack is a prudent approach. These libraries provide efficient encoding/decoding, support for schema evolution (versioning), and are generally more compact than text-based formats like JSON or XML.[24]

3.2. Bloom Filters for Read Optimization

A Bloom filter is a space-efficient, probabilistic data structure that provides a powerful optimization for read operations in a key-value store. It can answer set membership queries with a crucial asymmetry: it can definitively state that an element is *not* in the set (a "no" answer, which is always correct), but can only state that an element is *probably* in the set (a "yes" answer, which carries a configurable risk of being a false positive).[14]

Within our storage engine, one Bloom filter will be generated for each SSTable created. This filter is constructed by hashing every key in the SSTable k times and setting the corresponding bits in a bit array. The filter is then stored in the Filter Block of the SSTable file and is typically loaded into memory when the SSTable is opened.

The integration of the Bloom filter into the read path provides a significant performance enhancement, particularly for queries targeting non-existent keys—a common access pattern in many applications. Before the engine incurs the cost of a disk read to search an SSTable, it first queries the corresponding in-memory Bloom filter with the target key.

- If the Bloom filter returns "no," the engine is 100% certain the key does not exist in that SSTable and can immediately skip it, avoiding all associated I/O and CPU costs for that file.
- If the Bloom filter returns "yes," the key *may* be in the SSTable (or it could be a false positive). Only in this case does the engine proceed with reading the index and data blocks from the disk to confirm the key's presence.[2]

By filtering out unnecessary disk reads, Bloom filters dramatically reduce read latency and I/O load, especially in a deeply leveled LSM-Tree where a key lookup might otherwise need to check dozens of files. The trade-off is a modest amount of memory required to hold the filters for all active SSTables and a small computational overhead during SSTable creation.

3.3. The MANIFEST File: Tracking Database State

The logical state of an LSM-Tree database at any point in time is defined by its collection of SSTables and their organization into levels. The MANIFEST file is the authoritative, durable record of this state. It functions as an append-only log that chronicles every change to the database's structure, such as the creation of a new SSTable from a Memtable flush or the replacement of a set of old SSTables with new ones after a compaction.[27]

Each entry written to the MANIFEST describes a complete snapshot of the database's file layout, referred to as a Version. This Version object lists all live SSTable files for each level, their key ranges, and other relevant metadata.

Atomicity of state transitions is critical. A crash during a compaction operation, for instance, must not leave the database in an inconsistent state. This is achieved through an atomic rename mechanism. When the database state changes (e.g., a compaction completes), the engine writes the entire new Version state to a new, temporary MANIFEST file. Once this file is durably written to disk, an atomic rename() system call is used to replace the old CURRENT file (a simple text file containing the name of the active MANIFEST) to point to this new manifest. This atomic rename ensures that the switch to the new database state is an all-or-nothing operation. If the system crashes at any point, the CURRENT file will point to the last fully consistent MANIFEST, from which the database can safely recover its state.

Part II: Indexing and Concurrency

This part delves into the most innovative and technically demanding aspects of the storage engine: the design of a persistent, version-aware Trie for high-speed indexing and its seamless integration with a lock-free Multi-Version Concurrency Control (MVCC) framework. This combination is the key to achieving both high performance and strong transactional guarantees without traditional locking.

Section 4: A Persistent, Version-Aware Trie for High-Speed Indexing

The choice of indexing structure is a pivotal architectural decision. While B+Trees are the de facto standard for on-disk databases due to their excellent range scan performance and I/O characteristics, the user requirement for a Trie-based index presents an opportunity to optimize for different access patterns and leverage unique structural properties for a lock-free MVCC implementation.

4.1. Trie Variant Selection: Adaptive Radix Tree (ART)

A standard Trie, while simple, suffers from high memory consumption and poor cache locality due to its numerous pointers and nodes, especially for sparse key sets. A Radix Tree (also known as a Patricia Trie) improves upon this by compressing chains of nodes with only a single child into a single edge, reducing the tree's depth and memory footprint.[28] However, the

Adaptive Radix Tree (ART) represents a significant leap forward in efficiency. An ART optimizes space and cache performance by dynamically selecting the most appropriate node type based on the number of its children. For example, a node with four or fewer children uses a compact Node4 structure (a simple array of keys and pointers), while a node with up to 256 children uses a Node256 (a full 256-entry pointer array).[29]

This adaptive nature makes ARTs exceptionally space-efficient and cache-friendly. Furthermore, ARTs possess several properties that are highly desirable for our storage engine:

- **Key Length-Dependent Performance:** Lookup complexity is proportional to the length of the key, not the total number of keys in the index, making it very fast for point lookups.[8]
- **No Rebalancing:** Unlike B+Trees, ARTs do not require complex and costly rebalancing operations upon insertion or deletion. The structure of the tree is determined solely by the keys it contains, which simplifies concurrent modifications.[30]

The following table provides a comparative analysis to justify the selection of ART over the more conventional B+Tree for this specific engine design.

Feature	B+Tree	Adaptive Radix Tree (ART)
Point Lookup Complexity	O(logBN), where B is branching factor, N is # of keys.	O(K), where K is key length. Independent of N.
Range Scan Efficiency	Excellent. Leaf nodes are linked, allowing for highly efficient sequential scans.	Good. Can perform ordered traversal, but may be less I/O-efficient than B+Tree's linked leaves.
Space Usage	Generally efficient, but internal nodes can have low utilization (min 50%).	Highly space-efficient due to adaptive node sizes and path compression.
Cache Efficiency	Good. High branching factor keeps tree shallow. Nodes are block-sized.	Excellent. Compact nodes are designed to fit well within CPU cache lines.
Concurrency/Lock-Free Implementation	Complex. Requires sophisticated latching or locking protocols for node splits/merges.	More amenable. No rebalancing simplifies logic. Path copying for MVCC is natural.
Insert/Delete Complexity (Rebalancing)	Involves potentially cascading node splits and	No rebalancing required. Involves simple node

	merges, which are complex, locking-intensive operations.	creations or type promotions/demotions.

Given the engine's goals of ultra-fast lookups and a lock-free concurrency model, the ART's strengths in cache efficiency, key length-dependent performance, and simplified modification logic make it the superior choice over a B+Tree.

4.2. Designing a Persistent, Lazy-Loadable ART

Persisting a pointer-based in-memory data structure like an ART to disk requires a carefully designed on-disk format that supports efficient loading and minimizes memory overhead. We will adopt a sophisticated strategy pioneered by systems like DuckDB, which enables lazy loading of the index on demand.[8]

- **On-Disk Format via Post-Order Traversal:** The ART will be serialized to disk using a **post-order traversal**. In this method, all of a node's children are written to disk *before* the node itself. This is a critical design choice. When the parent node is finally serialized, it already knows the exact on-disk location (a <BlockID, Offset> pair) of each of its children. These physical disk pointers are then stored directly within the parent's on-disk representation. This structure is the key to enabling lazy loading: to traverse from a parent to a child, the engine reads the parent from disk, extracts the child's disk address, and can then fetch the child node directly without needing to load any other part of the index.
- **Pointer Swizzling for In-Memory Efficiency:** A naive implementation would require each in-memory ART node to store both a memory pointer (for fast traversal once loaded) and a disk location for each child, effectively doubling the pointer size and negating the ART's cache efficiency benefits. To solve this, we will use **pointer swizzling**.[8] Modern 64-bit architectures typically use only the lower 48 bits for virtual memory addressing, leaving the upper bits available. We will use the most significant bit of each 64-bit child pointer as a "swizzle flag":
    - **Flag is 0 (Unswizzled):** The pointer does not hold a memory address. Instead, the remaining 63 bits are used to encode the child node's on-disk location (BlockID and Offset). When the engine needs to traverse to this child, it reads the location, loads the node from disk into memory, and then "swizzles" the parent's pointer by replacing the disk location with the new memory address and setting the flag to 1.
    - **Flag is 1 (Swizzled):** The pointer holds a valid in-memory virtual address. The node has already been loaded, and the traversal can proceed at memory speed without any disk I/O.

This technique allows the in-memory ART nodes to retain their compact size while seamlessly managing the transition of nodes from on-disk to in-memory state on demand.

4.3. Integrating the ART with MVCC: Path Copying and Versioning

To support MVCC, the index structure must not be modified in-place, as this would corrupt the view of older, still-active transaction snapshots. The solution is to treat the ART as a **persistent data structure**, in the functional programming sense of the term, using a technique called **path copying**.[31]

When a write operation (e.g., inserting a new key) is performed, the existing ART is not altered. Instead, the engine creates a new, updated version of the tree. This is done by:

1. Traversing the existing tree to find the insertion point.
2. Creating a new leaf node for the new key.
3. As the traversal path is unwound back to the root, a copy is made of each node along that path. Each copied parent node is updated to point to its newly created child (or the next copied node up the path).
4. All nodes and subtrees that are *not* on this modification path are not copied; the new copied nodes simply point to these existing, unchanged parts of the tree.[30]

This process results in the creation of a new root node for the ART. This new root represents the state of the index *after* the modification. The original tree, accessible via the old root, remains completely intact and unchanged. Each transaction's set of writes thus generates a new logical root. The global state of the database maintains an atomic pointer to the latest *committed* root. A transaction's snapshot is defined by the root pointer that was active at the moment the transaction began. This elegant mechanism allows dozens or hundreds of different transactions to concurrently see their own distinct, consistent versions of the index without any need for locks or latches on the index structure itself.

This approach reveals a profound architectural synergy: the use of path copying renders the Persistent ART itself a log-structured entity. Just as new data is written append-only to SSTables, new index nodes created during path copying are written to new locations on disk; old nodes are never overwritten. This means the on-disk format for our data files can be designed to store both serialized key-value data and serialized ART nodes. Consequently, the compaction and garbage collection mechanisms can operate uniformly on both data and index components. When a version of the ART (identified by its root) is no longer visible to any active transaction, its unshared nodes become garbage. The compactor can identify and reclaim the space occupied by these obsolete index nodes during its regular cycle, just as it does for outdated data versions. This unified, append-only philosophy for both data and metadata simplifies the entire storage layer, reduces complexity, and creates a cohesive and powerful architectural foundation.

Section 5: The Concurrency Model: Lock-Free MVCC Implementation

The concurrency control model is the heart of the storage engine, enabling simultaneous access while providing strong transactional guarantees. Our implementation of Multi-Version Concurrency Control (MVCC) is designed to be lock-free on the primary read and write paths, using C++ atomic primitives to manage state and ensure correctness.

5.1. Core Principles: Snapshots, Transaction IDs, and Visibility

The MVCC model operates on three core concepts:

- **Transaction Snapshots:** When a transaction begins, it is assigned a unique, monotonically increasing 64-bit **Transaction ID (TxnID)**.[34] It also logically captures a *snapshot* of the database state at that instant. This snapshot is defined by two key pieces of information:
    1. The transaction's own TxnID.
    2. A list of all other TxnIDs that belong to transactions that are currently in-progress (i.e., have started but not yet committed or aborted).
    3. A reference to the root of the committed ART index that was active at the start of the transaction.
- **Versioning Data Records:** Every key-value pair stored in the engine, whether in the Memtable or an SSTable, is not merely a (key, value) tuple. It is a versioned record with the structure: (key, creation_TxnID, deletion_TxnID, value).
    - creation_TxnID: The TxnID of the transaction that created this version.
    - deletion_TxnID: Initially null (or a max value). When this version is deleted or updated, this field is set to the TxnID of the deleting/updating transaction. An update is logically treated as a deletion of the old version and an insertion of a new one.[34]
- **The Visibility Rule:** This rule is the cornerstone of MVCC and determines which version of a record is visible to a given transaction. A version of a key is visible to a transaction T if and only if all of the following conditions are met:
    1. The version's creation_TxnID is less than T's own TxnID. (The version was created by a transaction that started before T).
    2. The creation_TxnID does not appear in T's snapshot list of in-progress transactions. (The creating transaction had already committed when T began).

3. The deletion_TxnID is null, OR the deletion_TxnID is greater than T's own TxnID, OR the deletion_TxnID belongs to a transaction that was in T's snapshot list of in-progress transactions. (The version was not deleted, or was deleted by a transaction that had not yet committed when T began).

This set of rules precisely defines a consistent, point-in-time view of the database for each transaction, forming the basis for Snapshot Isolation.[36]

5.2. Lock-Free Primitives in C++

The implementation will rely heavily on the C++11/14/17 standard library's atomic operations to manage shared state without resorting to mutexes on performance-critical paths.

- **Transaction ID Generation:** A global, system-wide transaction ID counter will be implemented as an std::atomic<uint64_t>. New TxnIDs will be allocated using a single, lock-free fetch_add(1) instruction, which guarantees both uniqueness and strict monotonic ordering across all threads.[39]
- **Managing Shared State:** Critical global pointers, such as the pointer to the currently active Memtable and the pointer to the latest committed ART root, will be declared as std::atomic<T*>. All modifications to these pointers will be performed using a compare_exchange_strong (or compare_exchange_weak) operation inside a retry loop. This is the canonical pattern for lock-free updates to shared data: a thread reads the current value, computes the new value locally, and then attempts to atomically swap the shared pointer to its new value, but only if no other thread has changed it in the meantime.[16]
- **Memory Ordering:** Correctly using C++ memory ordering semantics is crucial for both correctness and performance. Overly strict ordering (like the default std::memory_order_seq_cst) can introduce unnecessary performance penalties, while overly relaxed ordering can lead to subtle data races. We will use a principled approach:
  - **Acquire-Release Semantics:** The publishing of a new shared state (e.g., swapping in a new Memtable or updating the committed ART root) will use std::memory_order_release. This ensures that all memory writes made by the publishing thread *before* the atomic store become visible to other threads. Conversely, threads that read this shared pointer will use std::memory_order_acquire, which ensures that they see all the memory writes that were made visible by the release operation. This acquire-release pairing establishes the necessary happens-before relationship for safe, lock-free communication.[20]
  - **Relaxed Ordering:** For operations that do not need to synchronize memory, such as simple counters or statistics, std::memory_order_relaxed can be used to minimize overhead.

5.3. The Read Path in Detail

A read operation for a key K within a transaction T is a non-blocking procedure that leverages the MVCC infrastructure:

1. The operation begins with the ART root pointer captured in transaction T's snapshot.
2. The engine traverses the ART index from this root. If a node along the path is not yet in memory, the pointer swizzling mechanism triggers a lazy load from disk.[8] This traversal is entirely read-only with respect to the snapshot's version of the tree.
3. The traversal leads to a leaf node corresponding to key K. This leaf node will not contain the value directly, but rather a pointer to the head of a *version chain* for that key. This version chain could start in the Memtable (for very recent data) or in an SSTable.
4. The engine then traverses this version chain (which is typically short). For each version encountered, it applies the visibility rule described in Section 5.1, comparing the version's creation_TxnID and deletion_TxnID against T's own TxnID and its snapshot of in-progress transactions.
5. The first version that satisfies the visibility rule is the correct one for transaction T.
6. If the visible version is marked with a "tombstone" (a special marker indicating deletion), the key is considered not to exist from the perspective of this transaction. If no visible version is found, the key also does not exist. Otherwise, the value from the visible version is returned.

This entire process involves no locks, allowing reads to proceed in parallel with each other and with concurrent write operations without any blocking.

Part III: Transactions and Maintenance

This part of the report details the integration of the previously described components to provide full ACID transactional guarantees. It also outlines the critical background processes—compaction and garbage collection—that are essential for maintaining the long-term performance, efficiency, and correctness of the storage engine.

Section 6: Achieving Full ACID Compliance

The ACID properties (Atomicity, Consistency, Isolation, Durability) are the bedrock of reliable transaction processing. Our architecture achieves these properties through a combination of the Write-Ahead Log (WAL), the Multi-Version Concurrency Control (MVCC) model, and a disciplined commit protocol.

6.1. Atomicity and Durability

Atomicity ensures that a transaction is an "all-or-nothing" proposition. **Durability** guarantees that once a transaction is committed, its effects are permanent and will survive system failures. Both of these properties are primarily upheld by the Write-Ahead Log.[42]

- **Atomicity:** A transaction may involve multiple Put and Delete operations. Each of these operations is logged to the WAL as it occurs. However, the transaction is not considered final until a special COMMIT_TX record, containing its unique TxnID, is successfully written to the log. If the system crashes at any point before this commit record is durably stored, the recovery process will see the transaction's preceding log entries but will not see the commit record. Consequently, all of the transaction's partial effects will be ignored, effectively rolling it back and ensuring atomicity.[45]
- **Durability:** The guarantee of durability is met at the precise moment the WAL buffer containing the COMMIT_TX record is successfully flushed to the physical storage medium via an fsync() system call. Once this call returns, the transaction's outcome is permanent. The engine can then acknowledge the commit to the client, confident that even an immediate power failure will not result in data loss, as the recovery process can replay the WAL to restore the committed state.[17]

6.2. Isolation through Snapshot Isolation

Isolation ensures that concurrent transactions do not interfere with one another, making it appear as if they are executing serially. The MVCC mechanism described in Part II directly implements a powerful isolation level known as **Snapshot Isolation (SI)**.[34]

Under SI, every transaction operates on a consistent snapshot of the database as it existed at the moment the transaction began. This is achieved through the visibility rule (Section 5.1), which filters the multiple versions of each data item to find the one appropriate for the transaction's snapshot. The key benefits of this approach are:

- **Readers Don't Block Writers:** A read transaction can access its snapshot of the data without acquiring any locks, so it will never be blocked by a concurrent write transaction that is creating new versions of the same data.
- **Writers Don't Block Readers:** A write transaction creates new versions of data items but does not modify the old versions that readers might be accessing. Therefore, writers do not block readers.

This non-blocking nature provides a very high degree of concurrency and effectively prevents common concurrency anomalies such as Dirty Reads, Non-Repeatable Reads, and Phantom Reads.[46]

6.3. Advanced Topic: Serializable Snapshot Isolation (SSI)

While Snapshot Isolation is robust, it is susceptible to a subtle but critical anomaly known as **write skew**. This occurs when two concurrent transactions, T1 and T2, read an overlapping set of data, make independent decisions based on that data, and then write to disjoint sets of data. Because their write sets do not overlap, they do not conflict under standard SI and both can commit. However, the outcome can be a state that would be impossible if T1 and T2 had executed in any serial order, thus violating the principle of serializability.[37]

To provide the strongest guarantee of **Serializability** (the "true" serializable isolation level), the engine's design can be extended to implement **Serializable Snapshot Isolation (SSI)**. SSI builds upon the foundation of SI by adding a conflict detection mechanism. In addition to their write sets, transactions must also track their read sets. The core idea is to detect dangerous "read-write" dependencies between concurrent transactions.

- **Conflict Detection:** When a transaction T1 is about to commit, the system checks for a specific dangerous pattern: it determines if any other transaction T2 that committed *after* T1 started but *before* T1 attempts to commit, wrote to any data items that T1 had previously read.
- **Aborting Transactions:** If such a read-write conflict is detected, it signifies a potential write skew anomaly. To prevent it, the system must abort one of the conflicting transactions (typically the one attempting to commit last). The aborted transaction can then be retried by the application.[46]

Implementing SSI adds computational overhead to the commit process due to the need to track and check read/write sets. However, it provides the highest level of isolation, allowing application developers to reason about transactions as if they execute one at a time, which greatly simplifies application logic.

6.4. Consistency

Consistency in the ACID model refers to the guarantee that any transaction will bring the database from one valid state to another. This property is a shared responsibility between the database engine and the application. The application defines the rules of a "valid state" (e.g., a bank account balance cannot be negative, an order must have a valid customer ID). The database engine's role is to provide the tools to enforce these rules. Our engine does this by ensuring the atomicity and isolation of transactions. If the database starts in a consistent state, and each transaction is logically correct (i.e., it transforms one consistent state into another), then the guarantees of atomicity and isolation ensure that even with high concurrency, the database will only ever transition between consistent states.[43]

Section 7: System Maintenance and Garbage Collection

A storage engine based on LSM-Trees and MVCC is not self-maintaining. Over time, the accumulation of deleted data and obsolete versions will degrade performance and consume excess storage. Two critical background processes, Compaction and Garbage Collection, are required to maintain the system's health and efficiency.

7.1. Compaction: Managing the LSM-Tree

Compaction is the process of merging on-disk SSTable files. It serves three primary purposes:

1. **Reduce Read Amplification:** By merging multiple smaller SSTables into fewer, larger ones, compaction reduces the number of files that must be consulted during a read operation.
2. **Reclaim Space:** It physically removes data that has been deleted or superseded by newer versions, freeing up disk space.
3. **Maintain Tree Structure:** It enforces the hierarchical structure of the LSM-Tree, where data gradually migrates from lower (newer, smaller) levels to higher (older, larger) levels.[2]

The choice of compaction strategy involves a fundamental trade-off between read amplification, write amplification (the total amount of data written to disk relative to the amount of user data written), and space amplification (the ratio of total disk space used to the logical size of the data).

- **Compaction Strategy: Leveled Compaction:** For a system designed for a balanced read/write workload, **Leveled Compaction** is the preferred strategy. This approach, used by LevelDB and as the default in RocksDB, organizes SSTables into distinct levels. A compaction operation typically involves selecting one file from Level-L and merging it with all files in Level-(L+1) that have an overlapping key range.[12] This strategy is particularly effective at minimizing space amplification, which is a critical concern when using expensive solid-state drives (SSDs).[52]

The following table summarizes the trade-offs between Leveled Compaction and the primary alternative, Size-Tiered Compaction.

Metric	Leveled Compaction	Size-Tiered Compaction
Read Amplification	Low. A key exists in at most one SSTable per level, limiting the number of files to check.	High. A key can exist in multiple SSTables within the same tier, requiring more lookups.
Write Amplification	High. Data is repeatedly rewritten as it moves down through the levels.	Low. Data is merged less frequently, with other files of similar size.
Space Amplification	Low. Tightly controls the total size of data, typically around 1.1x the live data size.	High. Can require significant free space for merging large, similarly-sized SSTables.
Compaction I/O Pattern	More frequent, smaller I/O operations.	Less frequent, but much larger and "spikier" I/O operations.

The analysis shows that while Size-Tiered Compaction offers lower write amplification, making it suitable for extremely write-heavy, append-only workloads, Leveled Compaction provides a better overall balance. Its superior control over read and space amplification makes it the more robust and predictable choice for the general-purpose, high-performance engine we are designing.[52]

7.2. Garbage Collection in an MVCC System

The MVCC architecture retains old versions of data to serve transactions operating on historical snapshots. These old versions become "garbage" once they are no longer visible to any currently active or future transaction.[56] Failure to collect this garbage leads to unbounded database growth and performance degradation as read operations must traverse increasingly long version chains. A long-running read transaction can effectively halt garbage collection, as it requires old versions to remain available.[57]

The garbage collection (GC) process is tightly integrated with the compaction mechanism:

1. **Determining the Low-Water Mark:** The Transaction Manager continuously tracks the state of all active transactions. The GC process requires a "low-water mark" timestamp, which is the TxnID of the oldest active transaction in the system. This is also known as the safePoint.[59] Any data version that was made obsolete (i.e., its deletion_TxnID was set) by a transaction that committed *before* this safePoint is guaranteed to be invisible to all current and future transactions and is therefore eligible for collection.
2. **Integrated Collection During Compaction:** The compactor provides the ideal context for performing garbage collection. As the compaction process reads input SSTables and merges their contents, it examines the version chain for each key. For each version in the chain, it checks if it is reclaimable. A version can be discarded if its deletion_TxnID is valid and is less than the current safePoint. Such versions are simply not written to the new output SSTable, and their space is thus reclaimed.[59]
3. **Garbage Collecting the Persistent ART:** The persistent ART, with its path-copying update mechanism, also generates garbage in the form of obsolete nodes that are no longer part of any active index version. Reclaiming this space requires a dedicated GC strategy for the index. A robust method is a periodic mark-and-sweep collection. The process starts with the roots of all currently active transaction snapshots and the latest committed root. It traverses the ART from these roots, marking all reachable nodes. Any node that remains unmarked after the traversal is unreachable and can be reclaimed. This index GC can be scheduled to run concurrently with data compaction, allowing for a holistic cleanup of both data and index structures.[30]

Part IV: Performance, Optimization, and Synthesis

This final part discusses practical considerations for performance tuning, advanced optimizations, and provides a concluding synthesis that ties the entire architectural design back to the initial project requirements.

Section 8: Performance Tuning and Advanced Optimizations

Achieving "ultra-fast" performance requires more than a sound architecture; it demands careful tuning of memory, I/O, and concurrency parameters to match the underlying hardware and specific workload characteristics.

8.1. Memory Management

Effective memory management is crucial for minimizing disk I/O, which is often the primary performance bottleneck.

- **Block Cache:** A significant portion of the available RAM will be allocated to a **block cache**, which stores recently used data blocks from SSTables. An LRU (Least Recently Used) eviction policy is a standard and effective choice for this cache.[11] When a read operation requires a data block that is not in the cache, it is fetched from disk and placed in the cache. Subsequent reads of data within that same block can then be served directly from memory, avoiding disk access entirely. The size of the block cache is one of the most critical tuning parameters for read performance.
- **Index Caching:** While the Persistent ART is designed to be lazy-loaded from disk, performance would suffer if every index traversal required disk reads. The pointer swizzling mechanism provides a natural and efficient form of index caching. When an ART node is loaded from disk, its parent's pointer is "swizzled" to point to the in-memory location. This node will then remain in memory, available for subsequent traversals at RAM speed, until it is evicted by a memory management policy (e.g., if memory pressure becomes high). Caching the upper levels of the ART, which are accessed most frequently, is particularly important for accelerating all lookups.

8.2. I/O Optimization

The engine's design is fundamentally shaped by the performance characteristics of modern storage devices.

- **Sequential vs. Random I/O:** The LSM-Tree architecture is inherently optimized for

sequential I/O on the write path. Appending to the WAL and flushing entire Memtables as new SSTables are sequential operations that can fully saturate the bandwidth of both HDDs and SSDs. Read operations, however, are inherently more **random**, involving lookups into the index followed by seeks to specific blocks within SSTables. The architecture mitigates the cost of random reads through several layers of optimization: the in-memory Memtable, the ART index for direct lookups, the Bloom filters to avoid unnecessary I/O, and the block cache to serve hot data from memory.

- **Direct I/O vs. Buffered I/O:** The engine can be configured to interact with the storage device in one of two modes. With **Buffered I/O**, all reads and writes go through the operating system's page cache. This is simpler to implement but can lead to "double caching"—where data exists once in the OS page cache and again in our application-level block cache, wasting memory. With **Direct I/O**, the engine bypasses the OS page cache and manages its own caching exclusively via the block cache. This approach provides more control and can eliminate the memory overhead of double caching, often leading to better and more predictable performance for a database, but it requires a more sophisticated and well-tuned internal cache. For a high-performance engine, providing the option for Direct I/O is essential.

8.3. Concurrency Tuning

The number of background threads allocated for maintenance tasks is a key lever for tuning the balance between write throughput and resource consumption.

- **Background Thread Pools:** Following the model of mature systems like RocksDB, it is beneficial to have separate thread pools for different background tasks.[62]
  - **Flush Threads:** A small pool of threads dedicated solely to flushing immutable Memtables to Level-0 SSTables. This is a high-priority task, as a backlog in the flush queue can eventually cause foreground writes to stall.[21]
  - Compaction Threads: A larger pool of threads for running the less urgent but computationally intensive compaction tasks between SSTable levels.
    The number of threads in each pool should be configurable to allow tuning based on the number of available CPU cores, the speed of the storage device, and the nature of the workload. Too few threads can create a bottleneck where data builds up in the upper levels of the LSM-Tree, increasing read amplification. Too many threads can cause excessive contention for CPU and I/O resources, impacting the latency of foreground user requests.4

Section 9: Synthesis and Final Architectural Review

This section provides a final review of the proposed architecture, demonstrating how its integrated components fulfill all the initial requirements, and considers potential avenues for future development.

9.1. Fulfilling the Requirements

The architectural design presented in this report systematically addresses each of the specified requirements for the storage engine:

- **No-lock Concurrency Support:** This is achieved through the core MVCC model, which isolates transactions in time rather than with locks. It is implemented using lock-free data structures like the concurrent Skip List for the Memtable and lock-free programming patterns (atomic operations, CAS loops) for managing all critical shared state pointers, such as the active Memtable and the committed Trie root.
- **MVCC and ACID Support:** The MVCC framework, with its versioned data records, transaction snapshots, and precise visibility rules, provides the foundation for Isolation. The Write-Ahead Log guarantees Atomicity and Durability. Together, these components enable full ACID compliance, with an optional extension to Serializable Snapshot Isolation for the strongest guarantees.
- **Key-Value Store Model with Trie Indexing:** The system is fundamentally a key-value store. It utilizes a highly efficient Adaptive Radix Tree as its primary index, satisfying the specific indexing requirement while providing excellent performance characteristics.
- **Custom Persistent File Storage:** The report defines detailed, custom binary formats for all on-disk structures, including the WAL, the block-based SSTables, the persistent ART nodes, and the MANIFEST file, all designed for performance and robustness.
- **Compaction Support:** A Leveled Compaction strategy is integrated as a core background process, essential for managing the LSM-Tree structure, reclaiming space, and maintaining read performance over time.
- **Ultra-Fast Reads and Writes:** High write throughput is achieved via the LSM-Tree's log-structured write path, group commit for the WAL, and pipelined Memtable flushing. Fast reads are a result of the efficient ART index, lazy-loading with pointer swizzling, the use of Bloom filters to eliminate unnecessary I/O, and a multi-layered caching system.
- **Use of Bloom Filters:** Bloom filters are a first-class component of the read path, integrated into each SSTable to provide a fast probabilistic check that dramatically reduces I/O for non-existent key lookups.

9.2. Future Work and Extensions

While the proposed design constitutes a complete and powerful single-node storage engine, several avenues for future extension exist:

- **Column Families:** The architecture could be extended to support Column Families, a concept popularized by RocksDB.[22] This would allow users to partition the key space into logical groups within a single database. Each column family could have its own Memtable and set of SSTables, enabling independent tuning of parameters like block size, compression algorithm, and compaction strategy to better suit the specific access patterns of different data types.
- **Distributed Architecture:** Although designed as a single-node engine, the core components are amenable to distribution. The key-value model could be sharded across multiple nodes using a consistent hashing scheme. Replication could be added by using a consensus algorithm like Raft to replicate the Write-Ahead Log across a group of nodes, forming the basis for a fault-tolerant, distributed database.
- **Advanced Compaction Strategies:** The choice of Leveled Compaction provides a strong baseline. For more specialized workloads, the engine could be extended to support other strategies. For instance, a hybrid "Tiered+Leveled" strategy could be implemented, using size-tiered compaction for the upper levels (to reduce write amplification on fresh data) and leveled compaction for the lower levels (to control space amplification for older data), offering a more dynamic and adaptive approach.[50]

Works cited

1. LevelDB - Database of Databases, accessed September 4, 2025, https://dbdb.io/db/leveldb
2. What Is a Log-Structured Merge Tree (LSM Tree)? - Aerospike, accessed September 4, 2025, https://aerospike.com/blog/log-structured-merge-tree-explained/
3. Log-structured merge-tree - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Log-structured_merge-tree
4. Performance data for LevelDB, Berkley DB and BangDB for Random Operations, accessed September 4, 2025, https://highscalability.com/performance-data-for-leveldb-berkley-db-and-bangdb-for-rando/
5. How RocksDB works - Artem Krylysov, accessed September 4, 2025, https://artem.krylysov.com/blog/2023/04/19/how-rocksdb-works/
6. Log structured merge (LSM) tree and Sorted string table (SST) | YugabyteDB Docs, accessed September 4, 2025, https://docs.yugabyte.com/preview/architecture/docdb/lsm-sst/
7. Why don't SQL DBs use Trie or Aho-Corasick automation to index string? - Quora, accessed September 4, 2025, https://www.quora.com/Why-dont-SQL-DBs-use-Trie-or-Aho-Corasick-automation-to-index-string

8. Persistent Storage of Adaptive Radix Trees in DuckDB – DuckDB, accessed September 4, 2025, https://duckdb.org/2022/07/27/art-storage.html
9. Trie - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Trie
10. 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
11. LevelDB - Ju Chen, accessed September 4, 2025, https://chenju2k6.github.io/blog/2018/11/leveldb
12. LevelDB - Riak Documentation, accessed September 4, 2025, https://docs.riak.com/riak/kv/latest/setup/planning/backend/leveldb/index.html
13. The Fundamentals of RocksDB - Stream, accessed September 4, 2025, https://getstream.io/blog/rocksdb-fundamentals/
14. Bloom filter | Docs - Redis, accessed September 4, 2025, https://redis.io/docs/latest/develop/data-types/probabilistic/bloom-filter/
15. Bloom filter - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Bloom_filter
16. 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/
17. Documentation: 17: 28.3. Write-Ahead Logging (WAL) - PostgreSQL, accessed September 4, 2025, https://www.postgresql.org/docs/current/wal-intro.html
18. LevelDB Explained - How To Read and Write WAL Logs, accessed September 4, 2025, https://selfboot.cn/en/2024/08/14/leveldb_source_wal_log/
19. 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
20. Applying Memory Model to Lock-Free Data Structures | CodeSignal Learn, accessed September 4, 2025, https://codesignal.com/learn/courses/lock-free-concurrent-data-structures/lessons/applying-memory-model-to-lock-free-data-structures
21. RocksDB Overview · facebook/rocksdb Wiki · GitHub, accessed September 4, 2025, https://github.com/facebook/rocksdb/wiki/RocksDB-Overview
22. Exploring RocksDB: A Key-Value Database for Efficient Data Management | by Ygor F., accessed September 4, 2025, https://medium.com/@ygordefraga/exploring-rocksdb-a-key-value-database-for-efficient-data-management-502a4c4d0a9f
23. Bloom filter indexes | Databricks on AWS, accessed September 4, 2025, https://docs.databricks.com/aws/en/optimizations/bloom-filters
24. Best practices for custom file structures - Stack Overflow, accessed September 4, 2025, https://stackoverflow.com/questions/600708/best-practices-for-custom-file-structures

25. The Architect's Guide to Data and File Formats - MinIO Blog, accessed September 4, 2025, https://blog.min.io/architects-guide-data-file-formats/
26. Implement fast, space-efficient lookups using Bloom filters in Amazon ElastiCache - AWS, accessed September 4, 2025, https://aws.amazon.com/blogs/database/implement-fast-space-efficient-lookups-using-bloom-filters-in-amazon-elasticache/
27. LevelDB (go version) architecture design analysis - go学习 - SegmentFault 思否, accessed September 4, 2025, https://segmentfault.com/a/1190000040286395/en
28. What is the difference between trie and radix trie data structures? - Stack Overflow, accessed September 4, 2025, https://stackoverflow.com/questions/14708134/what-is-the-difference-between-trie-and-radix-trie-data-structures
29. How I implemented an ART (Adaptive Radix Trie) data structure in Go to increase the performance of my database 2x : r/golang - Reddit, accessed September 4, 2025, https://www.reddit.com/r/golang/comments/ti6f42/how_i_implemented_an_art_adaptive_radix_trie_data/
30. 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
31. Persistent Trie | Set 1 (Introduction) - GeeksforGeeks, accessed September 4, 2025, https://www.geeksforgeeks.org/dsa/persistent-trie-set-1-introduction/
32. Persistent data structure - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Persistent_data_structure
33. How MVCC databases work internally | by Kousik Nath - Medium, accessed September 4, 2025, https://kousiknath.medium.com/how-mvcc-databases-work-internally-84a27a380283
34. Multiversion concurrency control - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Multiversion_concurrency_control
35. 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
36. 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/
37. Snapshot isolation - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Snapshot_isolation
38. Snapshot Isolation - Jepsen, accessed September 4, 2025, https://jepsen.io/consistency/models/snapshot-isolation
39. Non-blocking algorithm - Wikipedia, accessed September 4, 2025, https://en.wikipedia.org/wiki/Non-blocking_algorithm
40. Lock-free Programming in C++ with Herb Sutter - InfoQ, accessed September 4, 2025, https://www.infoq.com/news/2014/10/cpp-lock-free-programming/

41. Lockless Reader/Writer Synchronization In MVCC Implementation - Stack Overflow, accessed September 4, 2025, https://stackoverflow.com/questions/12383742/lockless-reader-writer-synchronization-in-mvcc-implementation
42. What are ACID Properties in Transactional Databases? - Airbyte, accessed September 4, 2025, https://airbyte.com/data-engineering-resources/transactional-databases-explained-acid-properties-and-best-practice
43. ACID Transactions in Databases: A Data Engineer's Guide - MotherDuck, accessed September 4, 2025, https://motherduck.com/learn-more/acid-transactions-sql/
44. How to Implement ACID Transactions in Your Database Applications - MoldStud, accessed September 4, 2025, https://moldstud.com/articles/p-how-to-implement-acid-transactions-in-your-database-applications-a-comprehensive-guide
45. In-Depth Guide on Database Transactions and ACID Principles | by AkashSDas | Medium, accessed September 4, 2025, https://medium.com/@akashsdas_dev/in-depth-guide-on-database-transactions-and-acid-principles-e2f7c19db3da
46. Serializable Snapshot Isolation in PostgreSQL, accessed September 4, 2025, https://www.eecs.umich.edu/courses/cse584/static_files/papers/snapshot-psql.pdf
47. Isolation Levels — part VI: Snapshot Isolation | by Franck Pachot - Medium, accessed September 4, 2025, https://franckpachot.medium.com/isolation-levels-part-vi-snapshot-isolation-3f6c1d3113d1
48. How to Implement Serializable Snapshot Isolation for Transactions - DEV Community, accessed September 4, 2025, https://dev.to/justlorain/how-to-implement-serializable-snapshot-isolation-for-transactions-4j38
49. ACID & Database Transactions 101: Keeping Data Consistent in Concurrent Systems, accessed September 4, 2025, https://www.designgurus.io/blog/acid-database-transaction
50. An In-depth Discussion on the LSM Compaction Mechanism - Alibaba Cloud Community, accessed September 4, 2025, https://www.alibabacloud.com/blog/an-in-depth-discussion-on-the-lsm-compaction-mechanism_596780
51. 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/
52. Universal Compaction · facebook/rocksdb Wiki - GitHub, accessed September 4, 2025, https://github.com/facebook/rocksdb/wiki/universal-compaction
53. When to Use Leveled Compaction in Cassandra - DataStax, accessed September 4, 2025, https://www.datastax.com/blog/when-use-leveled-compaction

54. Key Database Compaction Strategies Used In Distributed System, accessed September 4, 2025, https://balachandar-paulraj.medium.com/key-database-compaction-strategies-used-in-distributed-system-26f8976707bf
55. Is Leveled Compaction Strategy still beneficial for reads when Rows Are Write-Once?, accessed September 4, 2025, https://stackoverflow.com/questions/46814635/is-leveled-compaction-strategy-still-beneficial-for-reads-when-rows-are-write-on
56. pingcap.github.io, accessed September 4, 2025, https://pingcap.github.io/tidb-dev-guide/understand-tidb/mvcc-garbage-collection.html#:~:text=Semantically%2C%20a%20multi%2Dversion%20system,the%20impact%20on%20the%20system.
57. 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
58. Scalable garbage collection for in-memory MVCC systems | Request PDF - ResearchGate, accessed September 4, 2025, https://www.researchgate.net/publication/345254237_Scalable_garbage_collection_for_in-memory_MVCC_systems
59. MVCC garbage collection - TiDB Development Guide, accessed September 4, 2025, https://pingcap.github.io/tidb-dev-guide/understand-tidb/mvcc-garbage-collection.html
60. 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
61. 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.
62. Comparing LevelDB and RocksDB, take 2 - Small Datum, accessed September 4, 2025, http://smalldatum.blogspot.com/2015/04/comparing-leveldb-and-rocksdb-take-2.html
63. RocksDB Overview | TiDB Docs, accessed September 4, 2025, https://docs.pingcap.com/tidb/stable/rocksdb-overview/