SYSTEMS ENGINEERING PROJECT REPORT

Distributed Database Engine

Multi-Region Consensus • LSM-Tree Storage • ACID Transactions

Youdahe Asfaw

Computer Science • Gustavus Adolphus College
Distributed Systems • Infrastructure • Reliability Engineering

youdaheasfaw@gmail.com • linkedin.com/in/youdaheasfaw • youdahe.dev

Abstract

This report details the design and architecture of a fully distributed, horizontally scalable database engine inspired by systems such as CockroachDB and Google Spanner. The project encompasses a custom Log-Structured Merge Tree storage engine, Raft-based consensus for fault-tolerant replication, a SQL-compatible query layer with latency-aware multi-region routing, ACID transaction support via two-phase commit and optimistic concurrency control, and automatic resharding using consistent hashing. The system supports configurable consistency levels including strong, eventual, and causal consistency modes.

Table of Contents

1. System Architecture Overview

The distributed database engine is organized into five primary layers, each responsible for a distinct concern in the data lifecycle. This layered approach ensures clean separation of responsibilities while enabling tight integration where performance demands it.

The **client layer** accepts connections via a SQL-compatible wire protocol and a key-value API. The **query routing layer** parses queries, plans execution, and routes operations to the appropriate shard based on key ranges and consistency requirements. The **consensus layer** implements the Raft protocol per shard to ensure fault-tolerant replication. The **storage engine** persists data using a Log-Structured Merge Tree optimized for write-heavy workloads. Finally, the **sharding layer** distributes data across nodes and regions using consistent hashing with virtual nodes.

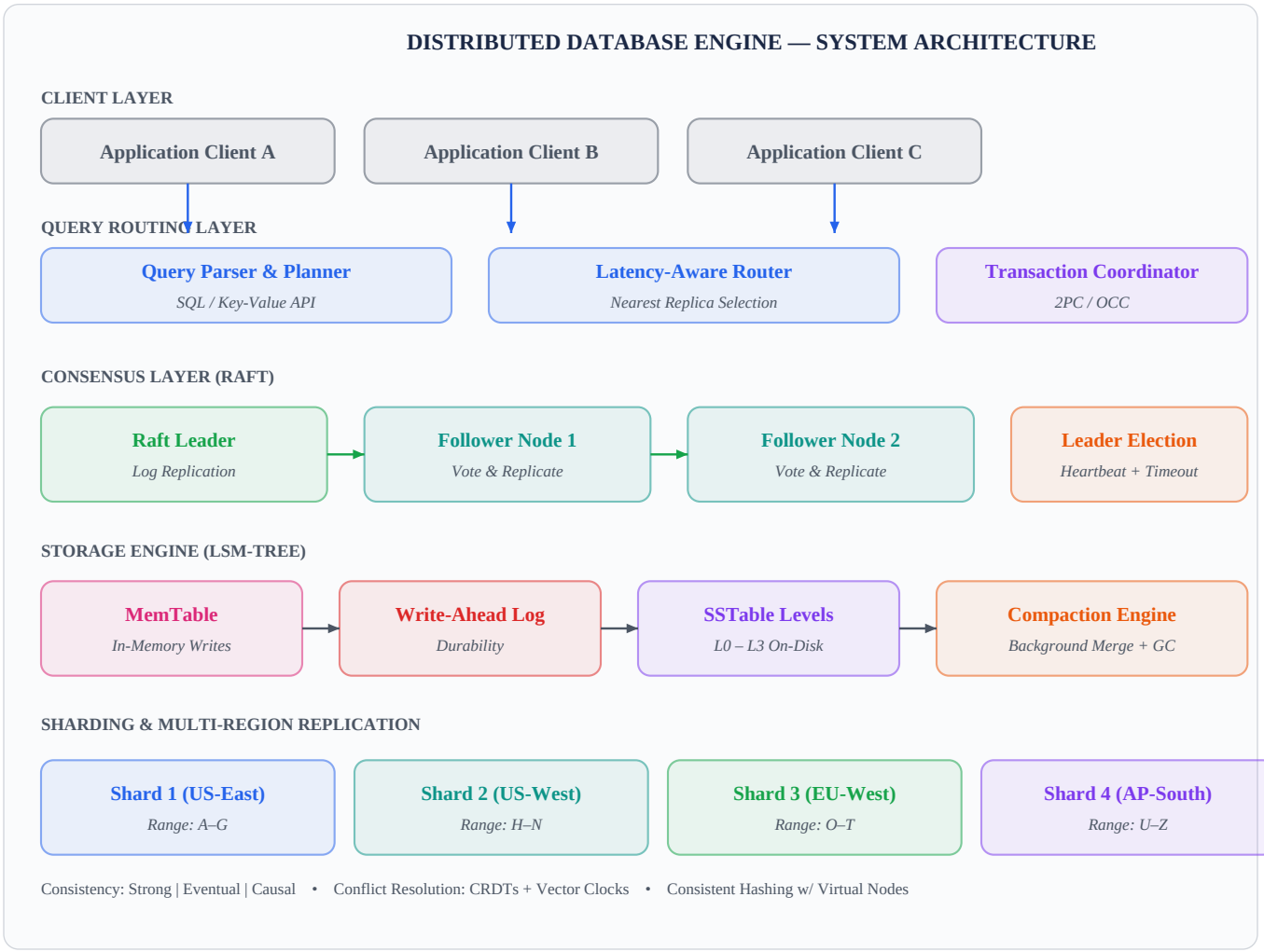

Figure 1 — Full system architecture showing all five layers and their interactions.

2. Storage Engine (LSM-Tree)

2.1 Write Path

Every write operation follows a strict durability-first path. The operation is first serialized and appended to the Write-Ahead Log (WAL), a sequential, append-only file on disk. Only after the WAL write is confirmed durable (via fsync) is the operation applied to the in-memory MemTable. The MemTable is implemented as a concurrent skip list, providing O(log n) insertions and lookups while supporting lock-free concurrent reads.

When the MemTable reaches a configurable size threshold (typically 64–128 MB), it is frozen and a new empty MemTable is created to accept incoming writes. The frozen MemTable is then flushed to disk as an immutable Sorted String Table (SSTable) at Level 0. Each SSTable contains a sorted sequence of key-value pairs, a Bloom filter for probabilistic membership testing, and an index block for binary search over data blocks.

2.2 Read Path

Point reads traverse the storage hierarchy from newest to oldest data. The active MemTable is checked first, then any frozen MemTable awaiting flush, then SSTables from Level 0 through Level 3. At each SSTable level, the Bloom filter is consulted before performing any disk I/O—with a false positive rate of approximately 1%, this eliminates the vast majority of unnecessary reads. Range scans use a merge iterator that maintains a priority queue across all levels, yielding keys in sorted order.

2.3 Compaction

Background compaction is essential for maintaining read performance as data accumulates. The system implements a leveled compaction strategy: when Level N exceeds its size target, selected SSTables are merged with overlapping SSTables at Level N+1. During compaction, tombstones (deletion markers) are garbage collected, duplicate keys are resolved to their latest version, and the resulting SSTables are written with fresh Bloom filters and indexes. Compaction is rate-limited to prevent I/O starvation of foreground operations.

3. Consensus Protocol (Raft)

Each shard (a contiguous range of keys) maintains its own independent Raft group consisting of a leader and two or more followers. This per-shard isolation ensures that consensus overhead scales with the number of active shards rather than total cluster size.

3.1 Leader Election

Followers maintain a randomized election timeout (typically 150–300ms). If a follower receives no heartbeat from the current leader within this window, it transitions to candidate status, increments the current term, votes for itself, and sends RequestVote RPCs to all peers. A candidate wins the election upon receiving votes from a majority of the group. The randomized timeout ensures that split votes are rare and resolve quickly through successive rounds.

3.2 Log Replication

The leader accepts all client writes, assigns each a monotonically increasing log index, and replicates the entry to followers via AppendEntries RPCs. A log entry is considered committed once a majority of nodes have

durably stored it. The leader then applies the committed entry to its state machine (the storage engine) and returns the result to the client. Followers apply committed entries in order, ensuring all replicas converge to the same state.

3.3 Log Compaction & Snapshots

Over time, the Raft log grows unboundedly. Periodic snapshotting captures the full state machine state at a given log index, allowing all preceding log entries to be discarded. When a new node joins or a lagging follower needs to catch up beyond the oldest available log entry, the leader transfers its latest snapshot followed by subsequent log entries. Joint consensus is used for safe membership changes—adding or removing nodes without risking split-brain scenarios.

4. Transaction Layer

4.1 Two-Phase Commit (2PC)

Cross-shard transactions require coordination across multiple Raft groups. The transaction coordinator implements the two-phase commit protocol: in the **prepare phase**, each participating shard acquires the necessary locks, validates constraints, and writes a prepare record to its WAL. If all shards vote to commit, the coordinator writes a commit decision record and sends commit messages to all participants (**commit phase**). If any shard votes to abort, the entire transaction is rolled back. The coordinator's decision record ensures crash recovery—on restart, in-doubt transactions are resolved by reading the coordinator's log.

4.2 Optimistic Concurrency Control (OCC)

For read-heavy workloads where contention is low, optimistic concurrency control avoids the overhead of acquiring locks upfront. Transactions read freely during their execution phase, recording the versions of all accessed keys. At commit time, a validation phase checks whether any of these keys have been modified by concurrent transactions since they were read. If validation passes, the writes are applied atomically; otherwise, the transaction is aborted and retried with fresh reads.

4.3 Hybrid Logical Clocks (HLC)

Causally consistent timestamps are essential for multi-region operation where physical clocks cannot be perfectly synchronized. Hybrid Logical Clocks combine a physical timestamp component with a logical counter. The physical component tracks wall-clock time (with bounded drift), while the logical component resolves ordering ambiguities when physical timestamps collide. This provides a total ordering of events that respects causality without requiring specialized hardware like GPS or atomic clocks.

5. Query Layer & Routing

The query layer accepts SQL statements through a PostgreSQL-compatible wire protocol. An LALR parser generates an abstract syntax tree, which is then transformed by the query planner into an optimized execution plan. The cost-based optimizer considers statistics about data distribution, index availability, and estimated

cardinalities to choose between scan strategies (full table scan vs. index scan), join algorithms (hash join vs. merge join), and execution ordering.

The latency-aware router maintains a continuously updated map of shard locations and their measured round-trip times. For queries requiring **strong consistency**, reads are routed to the Raft leader of the relevant shard. For **eventual consistency**, reads may be served by any replica, preferring the geographically nearest one. **Causal consistency** uses HLC timestamps to ensure reads reflect all causally preceding writes, allowing reads from followers that have caught up to the required timestamp.

6. Sharding & Multi-Region Replication

Data is distributed across shards using consistent hashing with 256 virtual nodes per physical node. This ensures that when a node joins or leaves the cluster, only a proportional fraction of keys are remapped. Each shard is replicated across a configurable number of regions (default: 3) with placement constraints—for example, requiring at least one replica in each of US-East, US-West, and EU-West.

Automatic rebalancing triggers when shard sizes diverge beyond a configurable threshold. The rebalancer splits oversized shards at their midpoint key and merges undersized adjacent shards. During rebalancing, the system maintains availability by serving reads from existing replicas while the new shard configuration is being replicated. Conflict resolution for eventually consistent replicas uses Conflict-free Replicated Data Types (CRDTs) for commutative operations and vector clocks for detecting and resolving conflicting writes.

7. Observability & Testing

The system exports Prometheus-compatible metrics at every layer: storage engine compaction rates and amplification factors, Raft leader election frequency and replication lag, query latency percentiles (p50, p95, p99), and transaction commit/abort ratios. Distributed tracing via OpenTelemetry propagates trace context through all layers, enabling end-to-end latency analysis from client request to storage engine response.

Correctness is verified through Jepsen-style chaos testing. The test harness injects network partitions, clock skew, process crashes, and disk failures while concurrent clients execute transactions. Post-test analysis verifies linearizability of committed operations, ensuring that no committed data is lost and no stale reads are served under strong consistency mode.

8. Technology Stack Summary

Component	Technology / Approach
Storage Engine	Custom LSM-Tree in Rust with Bloom filters, WAL, leveled compaction
Consensus	Raft (custom implementation) with log compaction and snapshots
Query Layer	LALR SQL parser, cost-based query planner and optimizer

Transactions	2PC coordinator, OCC, Hybrid Logical Clocks
Sharding	Consistent hashing with 256 virtual nodes, auto-rebalancing
Networking	gRPC for inter-node communication, PostgreSQL wire protocol for clients
Observability	Prometheus metrics, OpenTelemetry distributed tracing, Grafana dashboards
Testing	Jepsen-style chaos testing, property-based testing, fuzz testing
Languages	Rust (storage, consensus, networking), Python (testing, tooling)